FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Efficient Target and Application Specific Selection and Ordering of Compiler Passes

Ricardo Jorge Ferreira Nobre

Programa Doutoral em Engenharia Informática

Supervisor: Prof. Dr. João Manuel Paiva Cardoso

October 27, 2017

Efficient Target and Application Specific Selection and Ordering of Compiler Passes

Ricardo Jorge Ferreira Nobre

Programa Doutoral em Engenharia Informática

GRANT NUMBER: **SFRH/BD/82606/2011**

October 27, 2017

Resumo

Aquando da compilação de código fonte, os programadores usualmente utilizam uma das flags representativas de um dos níveis de otimização do compilador. Essas flags de compilação representam sequências fixas de passos de compilação, e, como tal, em algumas situações a utilização de sequências de compilação adaptadas ao código fonte e arquitetura alvo pode resultar em melhor desempenho e/ou na melhoria de outras métricas como tamanho de código. Por forma a maximizar o potencial de otimização possível através da exploração de sequencias de compilação, é importante não só selecionar que passos de compilação devem ser utilizados mas também a sua ordem de execução. Com o crescente número de passos de compilação de análise e de transformação disponibilizados pelos compiladores dos dias de hoje, a seleção de sequências de passos é uma tarefa desafiante, já que o espaço de exploração rapidamente se torna demasiado grande, tornando-se necessária a utilização de algoritmos de otimização dedicados a encontrar sequências de otimização próximas das ótimas e/ou heurísticas para reduzir o espaço de procura. Nesta tese de doutoramento é proposto um sistema modular que é capaz de explorar sequências de passos de compilação. Neste sistema é utilizada a linguagem de programação orientada a aspetos LARA, para implementar algoritmos/métodos de exploração do espaço de projeto (do inglês: Design Space Exploration, DSE) para explorar sequências de compilação, e, no caso de um dos compiladores suportados por este sistema, a linguagem LARA também pode ser utilizada para guiar internamente as ações do compilador em termos da aplicação de transformações específicas, nomeadamente na seleção dos pontos do programa que se quer otimizar e das transformações a aplicar. No âmbito do trabalho apresentamos soluções para DSE que investigámos e desenvolvemos. Nas experiencias apresentadas nesta tese tivemos maioritariamente como alvo processadores e benchmarks relacionados com sistemas embebidos. Utilizando as abordagens desenvolvidas, fomos capazes de obter de forma eficiente melhorias de otimização significativas de forma consistente comparando com a utilização dos níveis de otimização padrão existentes em compiladores atuais.

Abstract

Programmers usually rely on one from a set of optimizing compiler optimization level flags shipped with the compiler they are using to compile their source code. Those compiler flags represent fixed compiler pass sequences, and therefore in some situations better performance and/or other metrics such as code size can be achieved if using compiler sequences "tuned" for the specific source code and target architecture. In order to maximize the potential of optimization made available with compiler sequence exploration, it is required not only to select what compiler passes to execute, but also to select their order of execution. With the growing number of analysis and transformation compiler passes that are becoming available on modern compilers, the selection of compiler sequences can be a challenge, as the exploration space quickly becomes too large, thus requiring the use of optimization algorithms suited to the task of finding close to optimum compiler sequences and/or heuristics to prune the exploration space. In this thesis we propose a modular system that is capable of exploring compiler pass sequences. The system relies on LARA, an aspect oriented programming language, to implement Design Space Exploration (DSE) algorithms/methods for exploring compiler sequences, and additionally, in the case of one of the compiler toolchains supported by this system, LARA can be used to internally guide the actions of the compiler tools in relation to where to apply specific transformations in the program/application being optimized. We present DSE schemes that we researched and developed in the context of this work. In the experiments presented in this thesis we mainly targeted processors and benchmarks related with embedded systems. Using our approaches, we were able to efficiently and consistently achieve significant optimization improvements versus any the standard optimization levels available in current compilers.

Acknowledgments

I would like to thank my father, mother, sister and all my close relatives in general. They supported me in many ways conductive to this work. I would like to partially dedicate this thesis to my aunt Raquel who gave me access to a computer early on. A very special thank you to my mother because I believe she was the person with most positive influence in my development and always wanted my well being above all.

I would like to thank my supervisor, Prof. Dr. João M.P. Cardoso, for all the support he has been able to give me; and Prof. Dr. Pedro Diniz, from ISI, USA[1], who is possibly the one who most influenced me to pursue a Ph.D. and helped creating the opportunities for that to happen.

I acknowledge the support given by Tiago Carvalho, regarding the usage of the LARA tools. Tiago Carvalho is a Ph.D. student working next to me at SPeCS Lab[2]. I also acknowledge Pedro Pinto for the support given regarding a source-to-source compiler called MANET, and Luís Reis for giving a very important contribution for manually validating a number of NVIDIA PTX representations of OpenCL kernels generated after optimization.

I would like to acknowledge the support for this work of ACE Associated Compiler Experts by providing the complete license to use the CoSy compiler development system at FEUP, as well as the helpful insights about the CoSy compiler. I acknowledge Max Ferger, Bryan Oliver, Liam Fitzpatrick and André Vieira from ACE[3] for giving me support regarding the CoSy compiler development system; which I used to develop the REFLECTC compiler (Online DEMO[4]).

I acknowledge the help Gabriel Coutinho from Imperial College London[5] gave me with the C++ implementation of a LARA (an aspect-oriented language) weaving engine, which adds support for conditionally applying transformations over selected parts of the program code, for use with the REFLECTC compiler. I would like to express my gratitude to everyone that exchanged ideas with me about my Ph.D. work. Specifically, Ali Azarian (former FEUP Ph.D. student), Adriano Kaminski (FEUP Ph.D. student), André Santos (former Ph.D. student), André Fernandes (former FEUP researcher), Gonçalo Pereira (former FEUP researcher), João Bispo (FEUP Postdoctoral research assistant), Pedro Pinto (FEUP Ph.D. student), Luís Pedrosa (former FEUP M.Sc. student), Luiz Martins (former Ph.D. exchange student), Luís Reis (FEUP Ph.D. student), Maria Pedroto (FEUP Ph.D. student) and Nuno Paulino (former Ph.D. student). Additionally, I would like to thank Pedro Brandão (FEUP Ph.D. Student), Gustavo Laboreiro (FEUP Ph.D. student) for all the interesting conversations. A very special thank you to Rafaela de Faria and to Michael Tipton for reviewing late versions of this document.

I gratefully acknowledge the financial support provided by Fundação para a Ciência e a Tecnologia[6] (FCT) under grant number SFRH/BD/82606/2011. I also acknowledge support provided by the European Comission's Framework Programme 7 (FP7) under contract No. 248976 (REFLECT project) and by the TEC4Growth project, "NORTE-01-0145-FEDER-000020", financed by the North Portugal Regional Operational Programme (NORTE 2020), under the PORTUGAL 2020 Partnership Agreement, and through the European Regional Development Fund (ERDF).

Ricardo Jorge Ferreira Nobre

[1] ISI – Information Sciences Institute: http://www.isi.edu

[2] SPeCS – Special-Purpose Computing Systems, Languages and Tools: http://specs.fe.up.pt/

[3] ACE – Associated Compiler Experts: http://www.ace.nl/

[4] REFLECTC DEMO: http://specs.fe.up.pt/tools/reflectc/

[5] Imperial College London: http://www3.imperial.ac.uk/

[6] FCT – Fundação para a Ciência e a Tecnologia: http://www.fct.pt/

Contents

List of Figures

List of Tables

Acronyms

AOP	Aspect-Oriented Programming
ALU	Arithmetic Logic Unit
ASIC	Application-Specific Integrated Circuit
CCMIR	CoSy Common Medium Intermediate Representation
CISC	Complex Instruction Set Computer
CMOS	Complementary Metal-Oxide-Semiconductor
CDFG	Control Data Flow Graph
CoSy	CoSy COmpiler SYstem
CPU	Central Processing Unit
DSE	Design Space Exploration
DSL	Domain Specific Language
DSP	Digital Signal Processor
EDL	Engine Description Language
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit
GA	Genetic Algorithm
GCC	GNU Compiler Collection
GPP	General Purpose Processor
GPU	Graphics Processing Unit
HPC	High Performance Computing
IR	Intermediate Representation
ISA	Instruction Set Architecture
MIPS	Million Instructions Per Second
MMU	Memory Management Unit
NFR	Non-Functional Requirements
NOP	No Operation
PC	Personal Computer
RAM	Random Access Memory
ROM	Read-Only Memory
REFLECT	REndering FPGAs to MuLti-Core Embedded CompuTing
RISC	Reduced Instruction Set Computer
RPC	Remote Procedure Call
SA	Simulated Annealing
SIMD	Single Instruction, Multiple Data
SVM	Support Vector Machine
VHDL	VHSIC Hardware Description Language
VLIW	Very Long Instruction Word

Chapter 1

Introduction

Embedded computing systems are pervasive in our society, and they continue to grow in number and functionality. BCC Research reports that the worldwide market for embedded technology was $158 billion in 2015, and expects the market to reach $221 billion by 2021 (BCC Research, 2016). The same report also refers that while the embedded hardware contributes to most of the embedded systems market percentage; the embedded software is the fastest growing segment.

Embedded systems are computing systems embedded in electronic devices. They are more pervasive than desktop computers, with tens of embedded systems per household and motor vehicle and annual production number of billions of units (Vahid and Givargis, 2001).

Embedded systems are mounted on compact electronic circuit boards integrated into devices such as of smartphones, pagers, cell-phone base stations, portable music/video players, disc/network-based video-players, game consoles, digital cameras, camcorders, automatic teller machines, printers, scanners, photocopiers and televisions, wearable devices (e.g., smart watches/rings/bracelets, biosensors as part of clothes), complex electronics in automobiles and airplanes (e.g., inertial guidance systems, global positioning system receivers, electronic motor controllers, anti-lock braking system, electronic stability control, traction control, cruise control, automatic four-wheel drive), medical equipment (e.g., electronic stethoscope, positron emission tomography scanner, life-support system, medical diagnostic systems) and other everyday house appliances (e.g., microwave, smart oven, fridge, washing machine, dishwasher, lighting/climate/security/surveillance controllers, temperature control, fire-alarm systems). They are also part of desktop and laptop computers, in the form of hard disk controllers, mouse tracking system inside modern computer, modems, MPEG2 and H256 coders/decoders, fingerprint readers, and systems for refresh rate synchronization inside monitors such as the proprietary NVIDIA G-Synch (NVIDIA Corporation, 2015), which relies on a Field-Programmable Gate Array (FPGA) module and dedicated memory.

Embedded systems tend to be more tightly constrained in certain aspects than general purpose computers, such as workstations, servers, and laptops. Because of their specialization to specific tasks, miniaturization, energy efficiency and cost related requirements; they tend to have far less memory, less storage capacity, and/or less general purpose processing power. Specialization often results in computer systems targeting the execution of embedded software that are not only more

energy efficient, but that can also provide higher throughput than traditional microprocessors for specific tasks.

A number of embedded systems rely on Digital Signal Processors (DSPs), Application Specific Integrated Circuits (ASICs), Graphics Processing Units (GPUs), and/or hardware implemented on reconfigurable fabrics using FPGAs, instead of or in addition to microcontrollers/microprocessors. Custom designed chips, such as DSPs or softcores in reconfigurable fabrics, can be used in an embedded system as a measure to more efficiently reach real-time constrains required by the reactive environments where some of these systems operate.

A strong emphasis on optimization when writing functions/programs can be arguably more important when targeting embedded systems than when developing for traditional computers. An application can be written (or rewritten) so that it uses less energy, less memory and/or executes faster; all requirements typical of embedded systems. A number of particularities have to be simultaneously taken into account when developing software for an embedded system, so that the cost of developing software is kept at a minimum while still being able to satisfy these requirements.

Embedded systems can have requirements related to size, which can be accomplished by using a small battery and/or hardware that dissipates less power, and can benefit from and be made to have increased performance or reliability (e.g., handle higher temperatures). They are often used in reactive environments with a high degree of unpredictability (e.g., navigation system) or in a context with well-defined variables (e.g., keyboard controller), and are often required to operate in real time, i.e., to compute with a maximum predictable delay.

Applications are typically programmed using a high-level language such as C or C++, instead of directly giving the microprocessor/microcontroller, DSP, or GPU, a list of the instructions to execute and their operands (i.e., machine language), and they can be improved in two orthogonal ways: (a) relying on transformations manually made by programmers at the source-code level in an effort to specialize the code to target a specific platform while satisfying the application non-functional requirements; (b) and/or relying on a tool, called a *compiler*, which will automatically apply transformations to the application code or, as it is more often the case, to a tool-dependent internal representation of the input code. The final step of a compilation process is the generation of a new representation of the application in other language, typically machine code that can be directly executed on a target architecture/device. Figure 1.1 depicts the inputs and outputs of an offline compiler, and the inputs and outputs of the resulting executable.

The selection of what compiler to use can have a considerable influence on a project (e.g., how fast the requirements are met). In the perspective of its users, what makes a compiler "better" is how consistently and how far it goes in improving their applications regarding the target hardware platforms and metric(s). Success depends on the target platform, the application itself, and on the transformations that are applied to a representation of the application, and their order of execution, during compilation (Torczon and Cooper, 2011). This thesis is concerned with the later, and to some extent, with its interaction with the others. It is concerned with efficient techniques to select and define the order of execution of compiler transformations so that the metrics (i.e., performance, code size, energy, power, size) that would influence the non-recurring and the unit costs can be

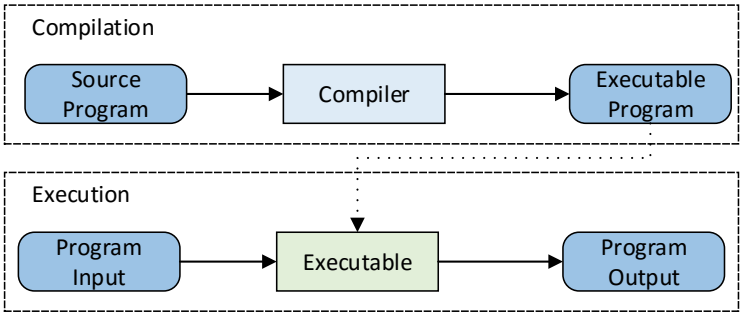

Figure 1.1: Offline compilation and execution of a program.

further improved compared with what is achieved relying only on the standard approaches.

Using efficient approaches to select and/or reorder compiler phases can result in reducing non-recurring costs, as suitable application implementations may be achieved faster and/or without spending as much engineering resources. In addition, specializing compiler phase selection and ordering when compiling embedded software can make it possible to use cheaper hardware, thus reducing the unit cost. Additionally, in some scenarios, in may only be required to maintain a simpler version of the source code. With suitable selection and ordering of compiler phases a larger portion of the optimization work can be relegated to the compiler. If developers decide to change the hardware used in a given embedded system (e.g., new Central Processing Unit (CPU) with radically different architecture) in a newer version of their product, their work retargeting the software can be reduced in relation with situations where they would have to deal with code that had been stringently hand-optimized to the previous hardware. Simpler code tends to be easier to retarget, therefore being able to use simpler code by relying more on the compiler toolchain ability to efficiently search for and/or suggest specialized compiler pass selections and/or orders can be of paramount importance.

We present in the next sections the motivation behind the work accomplished during this Ph.D. thesis; the problem definition; the hypothesis; the research questions; the impact and objectives of the work; our contributions; and an overview of the structure of this thesis.

1.1 Motivation

We present next, in more detail, the motivation behind the work we are pursuing, i.e., the key factors that make the search for methodologies and techniques for efficient compiler sequences exploration an interesting and relevant topic.

Mapping applications efficiently is very important when targeting systems with strict requirements, such as the ones existing in embedded systems and High Performance Computing (HPC) domains. What constitutes a suitable mapping strategy depends on a combination of concerns specific to the application, such as energy/power, performance, memory and/or storage. Software optimization driven by an optimizing compiler helps to comply with requirements while using

less resources in the process, contributing to the reduction of hardware costs and/or improving user experience.

Compilers often use heuristics to select optimization parameters and/or to decide to apply or not to apply a given transformation at the different locations of a function or an application. The reliance in the optimization levels typically distributed with the compiler toolchains (e.g., GCC –O3 (Free Software Foundation, b)) tends to result in solutions that represent an improvement over non-optimized code, but these solutions are in a number of cases very sub-optimal as these default optimization levels represent fixed compiler sequences. There are compiler toolchains, such as Clang/LLVM (LLVM Developer Group, a), that allow defining the order in which compiler passes are applied when compiling a given program/function. However, at the present time this feature is rarely used its users. Partially because of the fact that function-, target-, and metric-specific selection and ordering of compiler phases can result in compiled code with considerably higher quality over only relying on the standard optimization levels does not seems to be widely disseminated, and additionally because it is considered difficult to find suitable compiler sequences. Optimal code generation is in general considered NP-hard even with the unreasonable assumption that code phases are perfectly separated (Leupers, 1997). There is also the challenge of validating specialized compiler phase selections/orders, but the problems that can emerge from using compiler sequences different from the ones represented by the –Ox[1] flags can typically be mitigated in a number of scenarios. Additionally, we believe it is reasonable to assume that if interest in the use of compiler phase ordering specialization grows, then the current compiler toolchains will be adapted to better support this feature in a user friendlier way and the problems that can otherwise arise from executing the compiler phases in orders that were not stringently tested by the compiler developers can be mitigated by better verification of the correctness of the individual compiler passes.

Compiler phases can be mutually dependent. They can have positive or negative interactions and because of the large problem space (compilers have tens or hundreds of compiler passes), given an optimization metric (e.g., performance) it is not clear what is the best phase order. This challenge is known as the *phase ordering* problem (Touati and Barthou, 2006; Kulkarni and Cavazos, 2012; Jantz and Kulkarni, 2013a).

Even if the phase orders consider only a small set of compiler phases, manually devising suitable phase orderings requires deep knowledge about correlation between code features, target architecture and compiler passes interdependence. The particular compiler passes used in a compilation sequence and their order of execution can have a noticeable impact on the code that is generated after optimization. Which compiler passes and order of execution should be used depends on the particular source code, including the algorithm being implemented and the programming style, the non-functional requirements of the application being compiled, and the target hardware/software platform. Cooper et al. (2002) argue for the use of adaptive compilers, represented in Figure 1.2. These are compilers that are able to automatically tailor compiler pass

[1] here –Ox represents the possible optimization options that are typically provided by compilers as –O1, –O2, –O3, etc.

sequences to the source code being compiled and to a given objective function, based on information passed in a feedback loop.

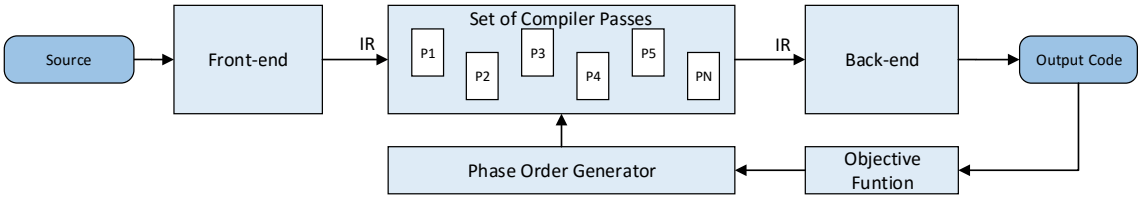

Figure 1.2: Adaptive compiler. Adapted from (Cooper et al., 2002).

1.1.1 Differences between source code domains and coding styles

While it may make sense to apply a given transformation (or set of transformations) on the IR generated by a compiler front-end from a given particular source code of a function/program, the same transformation(s) may not result in a desirable outcome when applied to other function/program. Or even an alternative source code representation of the same function/program with the same functional behavior. For instance, applying loop interchange (Torczon and Cooper, 2011) to the two innermost loops of a given function may yield more cache friendly memory access patterns, depending on the processor cache hierarchy, size, and cache policy; while applied to other function it may have the opposite effect. A matrix multiplication function implementing the simple $O(n^3)$ algorithm is an example that can often be significantly optimized with loop interchange, by decreasing cache misses. As an example, the study presented in (Nobre et al., 2014b) explores the use of different loop related transformations on a matrix multiplication kernel when targeting GPUs.

Other loop transformations, such as loop tiling and loop unrolling, can also result in better or worse performance depending on the particular program source code. For instance, performance will be worse if the loop code (or the size of the tiles) after unrolling (or tiling) is "too big" for the instruction caches. Thus, programming style can have a strong effect on the efficacy of application of automatic code optimization schemes.

1.1.2 Differences between target architectures

Target platforms can differ in relation to a number of variables. For instance, different families of microprocessors and microcontrollers have differences in respect to the type and number of instructions supported, the latency needed to execute each specific supported instruction, including the latency to fetch and decode instructions, how much energy is used, memory architecture and size, how many cycles for an instruction to be executed (i.e., depth) and how many instructions can be executed simultaneously (i.e., width) of the microprocessor/microcontroller pipeline. More specialized hardware displays even more profound architectural differences; such as GPUs, traditionally designed to accelerate the creation of images in a frame buffer intended for output

to a display; DSPs, used for digital signal processing tasks; and other accelerators; specifically accelerations relying on reconfigurable fabrics, such as FPGAs. Additionally, compilation does not only serve the purpose of generating software representations that will be executed on a microprocessor/microcontroller or other more specialized hardware. There are compilers that can generate representations of hardware components on the form of a Hardware Description Language (HDL), such as Verilog (see, e.g., LegUp (Canis et al., 2013)) or VHDL (see, e.g., DWARV 2.0 (Nane et al., 2012)). The hardware described in Verilog or VHDL can then be implemented in an FPGA or as an ASIC. In addition to being a function of the source of the application and the non-functional requirements, the improvement (e.g., speedup vs. a reference implementation) resulting from applying a given optimizing compiler sequence can differ when compiling to different target architectures. The target instruction set, how is each instruction implemented in hardware, memory configurations in relation to hierarchy and size, pipeline length and width, will result in differences in the configuration of the suitable compiler sequences for a given application and objective metrics. A number of researchers have shown that the compiler optimization selection and their parameters will depend largely on the target architecture (see, e.g., (Bodin et al., 1998)).

1.1.3 Differences between compilers and compiler releases

As a number of new compiler passes are added to a compiler toolchain during its lifetime (i.e., from the first to the most current version), the intuition that resulted in the –Ox sequences for the older releases can start to fall short. Additionally, new compiler passes tend to be considered to be added to the compiler optimizer in places where the likelihood of causing trouble (e.g., generating wrong code, crashing the compiler) are small. This can leave out a lot of potential for further optimization.

Even considerably small changes between different versions of the same compiler can lead to changes in the interdependencies between compiler passes. They can have an appreciable impact in the compiler pass phase selection and/or phase order configurations that are suitable given an application, a target platform and a set of requirements. The changes needed on specific compiler sequences found for a compiler version in order to adapt them to a new compiler version are potentially too difficult to grasp completely. They can be even impossible to grasp when there is no discernible pattern between the sequences found for the different compiler versions; making a new compiler pass selection/order exploration process needed.

Given the fact that the positive/negative interactions between compiler passes can change significantly between compiler versions, phase ordering can even be useful to determine new phase sequences to be represented by the –Ox optimization level flags (e.g., Clang/LLVM –O3).

1.1.4 Differences between non-functional requirements

Suitable compiler sequences for a given application/program are partially a function of the non-functional requirements of the application. A given compiler sequence that suits the compilation of a given source code in the context of a set of requirements may not produce the most desirable

output code in the context of other requirements. For instance, using a compiler pass that performs function inlining, a transformation that replaces function calls by the body of the function, may result in faster output code; but at the cost of a considerably larger code size, which may not be acceptable depending on the particular embedded hardware used. Embedded systems are often restricted by smaller memories. Furthermore, not only particular compiler passes will affect the quality of the generated output code in relation to the given metrics, but also compiler passes can interact with other compiler passes in ways that can affect the satisfaction of those metrics.

1.1.5 Current situation with hardware

We are currently in an era of diminishing returns in the context of performance of general purpose microprocessors/microcontrollers. While in the past one could almost count with a $2\times$ faster CPU in a couple of years and programmers could in a way rationalize not optimizing their software by convincing themselves that the problem would be timely solved by faster hardware, nowadays the situation is different. While there was a $1,000$-fold increase in microprocessor performance between 1985 and 2005 driven by improvements in transistor density, speed, and energy (Olukotun and Hammond, 2005), CPUs are reaching upper bounds in terms of what is known to be physically attainable with CMOS technology. While dramatic improvements in transistor density, speed, and energy were made possible by improvements in semiconductor fabrication and processor implementation, that combined with microarchitecture and memory-hierarchy techniques delivered 1,000-fold microprocessor performance improvement, this has not been the case in the last decade; even if considering the shift from singlecore to multicore architectures. Figure 1.3 depicts the evolution of CPUs from 1975 up to recently. Not only the single threaded performance in representative benchmarks is not improving as fast as before from around 2005, but also the number of cores in general-purpose CPUs (e.g., AMD/Intel x86) has remained constant for the most part until recently.

This makes improvements on the software side an even more important differentiation factor. Compiler phase ordering specialization is an orthogonal approach to other software based approaches that can be useful in situations that call for efficiency in using the available hardware.

Specialized electronic circuits, typically used in the context of computation for overcoming processing bottlenecks or for executing particular types of functions withing a computing system, such as GPUs, DSPs, and FPGAs, have been able to better translate increases in transistor density, transistor count and microarchitecture improvements to increases in performance and/or energy consumption reduction. Still, this does not make it any less important to explore compiler phase selection and ordering as an orthogonal approach to software optimization when targeting these computing architectures. In fact, their faster evolution may present even more opportunities to "outsmart" the default optimization options of the compilers, which may not been as well tuned to those targets when compared with compilers targeting the CPUs typically seen in desktops/laptops or servers.

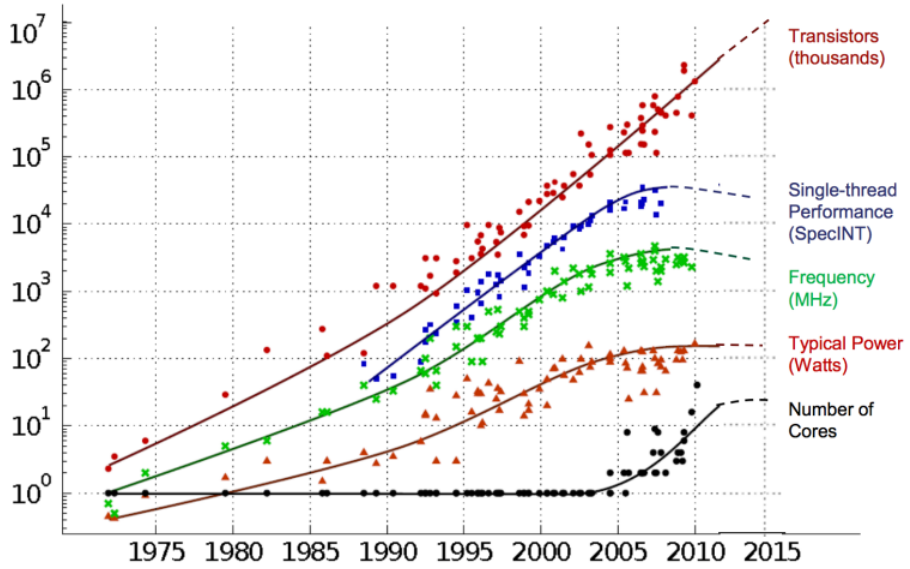

Figure 1.3: Number of transistors, single-thread performance, frequency, typical power and number of cores of CPUs over the years. Reprinted from (McGuire, 2014).

Preliminary experiments using automatic phase selection and ordering targeting OpenCL (Khronos, 2015) kernels to a NVIDIA GPU have been shown to result in considerable executable performance improvements, up to 5.7× and an average of 1.65×, when compared with the use of the default optimization options with the default compilation approach when targeting NVIDIA GPUs (i.e., online compilation); beating also CUDA (Harris, 2008) implementations (historically faster than OpenCL on NVIDIA hardware) of the same kernels by similar factors (Nobre et al., 2017). These results are also relevant in the context of embedded systems because the architectures used in embedded GPUs can rely on building blocks similar with those used in their desktop/laptop and server counterparts. For instance, the NVIDIA Tegra X2 system-on-a-chip features a custom ARM CPU and a NVIDIA Pascal-based GPU, the microarchitecture of the desktop GPU that was shown to benefit considerably with phase ordering specialization.

1.1.6 Special relevance to embedded systems

Embedded systems tend to be particularly more heterogeneous in regards to CPUs than desktop computers, workstations or laptops; which for the most part, have been using x86 CPUs; which although improved from generation to generation, tend to share many microarchitecture characteristics across the whole spectrum. The x86 CPUs from high-end workstations or even supercomputers are very similar to the CPUs in desktops and laptops, albeit having more cache, a larger number of cores, larger vector units, and in some cases a number of special instructions. This is not so much the case with embedded systems, with a number of different families of microprocessors/microcontrollers, DSPs and even reconfigurable fabrics in the form of, for instance, FPGAs. Adaptive compilation, i.e., compilation capable of producing executable code specially

tailored for a given target, has been suggested as a portable way to adapt for the myriad of different hardware targets characteristic of the Internet era (Diniz and Rinard, 1997).

A number of embedded systems execute performance-critical applications and/or are required to be able to adapt to a changing environment. This adds another dimension to the challenge of efficiently targeting embedded systems. There are a number of possible approaches to ensure efficient adaptation. One is in the way programs are developed, including rules for changing behavior depending on some input variables. For instance, the quality of the output output of a real-time video encoder may be programmed to change depending on the remaining battery capacity of a smartphone, or as it is more common, depending on the Internet connection when videoconferencing. Another approach, orthogonal to the previous one, is to create different versions of the encoder, and to select one during the application execution. In the latter case, different solutions (e.g., executable code for a CPU) can even be generated from the same source code with the use of different compiler phase selections and/or orderings. Each phase selection/order specializing the function/program, being targeted to a given embedded system, to a given use case (e.g., energy consumption prioritized, performance prioritized, a number of balanced modes with different ratios of importance for each metric).

Finding compiler sequences for a given application and target pair that result in better output code regarding the given metric(s) than the standard compiler sequences can be very useful. Especially in the embedded systems domain, were the code is compiled into a product that is potentially used by thousands or millions of users and cannot in some cases be updated. For instance, achieving higher performance than when using the standard compiler sequences may allow saving on the hardware costs of a product, resulting in cost savings that can be used to lower the price-point of the product or contribute to higher profits. In the same way, if a given compiler sequence allows generating code that uses less energy when executed on a given embedded system hardware, then the manufacturer can use a smaller/cheaper battery; and in the case a smaller program is generated, a smaller/cheaper ROM chip can be used. Finally, from the standpoint of the final consumer of the embedded systems, it can allow for a better user experience.

1.1.7 Relevance to other domains

Orthogonal approaches to optimization, such as compiler phase selection/ordering, can also have a positive impact in other domains, such as personal computing and HPC.

Desktop computers, laptops, tablets, smartphones are all types of personal computers used for consumption of information and/or for content creation. They all execute software that falls in the category of software that is deployed once and executed multiple times by numerous users, making this software a suitable candidate for optimization through compiler phase selection/ordering specialization.

HPC systems can offer petaflops of performance by relying on increasingly more heterogeneous systems, such as the combination of CPUs with accelerators in the form of GPUs programmed with languages such as CUDA (Harris, 2008) or OpenCL (Khronos, 2015). Heterogeneous systems are widespread in the HPC domain as a way to achieve energy efficiency and/or

performance levels that are not achievable by a single device/architecture (e.g., matrix multiplication is much faster in GPUs than in CPUs for the same power/energy budget (Betkaoui et al., 2010)).

A large number of specialized cores are offered in the form of accelerators that CPUs can use to offload computation that exhibits data-parallelism and often other types of parallelism as well (e.g., task-level parallelism). This adds an extra layer of complexity if one wants to target these systems efficiently, which in the case of HPC systems such as supercomputers is of utmost importance. An inefficient use of the hardware is amplified by the magnitude of such systems (hundreds/thousands of CPU cores and accelerators), with increasing utilization/power bill and/or cooling challenges as a consequence. Ensuring that the hardware is efficiently used is in part the responsibility of the compiler used (e.g., GCC and LLVM) to target the code to the computing devices in such systems and also the responsibility of the compiler users. The programmer(s) and the compiler(s) have to be able to target different computing devices (e.g., CPU, GPU, and/or FPGA) and/or architectures (e.g., heterogeneous system with ARM and x86 CPUs) in a manner that achieves suitable results for certain metrics, such as execution time and energy efficiency. Compiler phase selection/ordering specialization can be performed, for instance, after manually optimizing the source code to the point of diminishing returns.

1.2 Objectives

We present this Ph.D.'s problem definition, research hypothesis and research questions in the following subsections.

1.2.1 Problem definition

There is an increasing relevance of embedded systems (e.g., car onboard computers, smart glasses, smart watches, wearable computers) with their different processors/microcontrollers, homogeneous and heterogeneous multicore computing platforms (e.g., traditional computers, mobile devices such as tablets/smartphones, game consoles, high performance computing systems), general purpose computing on graphics processing units, and reconfigurable devices such as FPGAs. Techniques and tools for efficiently targeting one of these devices in the smallest amount of time (i.e., less human/hardware resources and/or time), are very desirable and therefore a hot topic in computer science, particularity in the compiler research domain. The problem this thesis addresses can be formulated as follows:

> "Given the source code of an application, a compiler, and a target platform in the form of a system with a microcontroller/microprocessor or GPU, how to find compiler sequences in the most efficient way so that the generated executable code better performs in relation to a given metric (e.g., performance, energy, code size)?"

1.2.2 Hypothesis

We suspect that it is possible to efficiently target a given application to different computing devices (e.g., CPU, GPU, FPGA) using not only the same phase selection/ordering exploration approach but also the same compilation flow in an integrated hardware/software compiler toolchain. In the sense that the mapping strategy derived using our technology and tools will be able to achieve implementation solutions (e.g., code for a microprocessor/microcontroller or a GPU, Verilog/VHDL for custom hardware implementations) competitive, in relation to the desired metrics, with solutions achieved with state-of-art approaches.

1.2.3 Research questions

Relevant research questions needed to be answered in the context of this Ph.D., as a direct or indirect result of the work done to achieve and validate our goals. Questions whose answers were researched and often addressed by experiments performed in the context of this Ph.D. include the following.

- *Q1* — How much can compiler sequences specialized for an application/program/code impact a chosen optimization metric on a given target architecture?

- *Q2* — Do different target architectures benefit differently from exploring specialization of compilation sequences as means to optimize a given function/program?

- *Q3* — What is the impact of the same compiler sequences across different target architectures?

- *Q4* — Can compiler sequence specialization be used, considering conceptual and practical reasons, as a means to optimize all types of programs/functions?

- *Q5* — What is the overhead in terms of number of compilations/executions caused by compiler pass sequence exploration on current state-of-art systems/algorithms?

- *Q6* — Are there modular frameworks that allow a given program/application to be compiled with support for compiler pass execution ordering specialization, with as little modifications as possible to the source code, target different computing devices, including CPUs, GPUs, FPGAs, other accelerator chips, while relying on a unified and integrated view?

- *Q7* — How much can we reduce DSE overhead continuing to achieve suitable phase orders?

- *Q8* — How effective and efficient we can be when recommending a small set of phase orders?

1.2.4 Explaining the research questions

In this section we explain in greater detail the research questions that we formulated at the start and during the Ph.D.

Question 1. Question *Q1* asks what is the potential benefit of tuning compiler sequences for a given function/program, regarding a given metric and a given target. We define the benefit of using compiler sequence specialization as the factor of improvement in relation to compilation of the same functions/programs using the traditional compiler optimization levels. For instance, the benefit of using compiler sequence specialization when striving to improve binary execution performance would be the performance improvement when executing a given function/program compiled with a suitable specialized compiler sequence over the same function/program compiled with the most aggressive optimization level (e.g., Clang/LLVM –O3). Results published by a number of authors (e.g., (Almagor et al., 2004; Kulkarni et al., 2010; Huang et al., 2015)) give strength to the argument that states that researching and developing more efficient technologies and tools for automatic tuning of a function/program to a given target computing device/platform by relying on individually tuning compiler sequences can be a worthwhile endeavor. This is especially the case when even small performance improvements, energy use reduction, or reduction in code size, can result in a reduction of the price to market or a better overall product in terms of user experience. A considerable percentage of the results published target outdated CPU architectures, use experimental compilers that are not representative of production compilers and/or were produced using old versions of production compilers; making it more challenging to compare with/between approaches from the state-of-art. Compilers evolve, both in the number of compiler passes available (e.g., LLVM 3.3 has 157 passes, LLVM 3.9 has 245 passes) and what each one of them does in terms of the actual work over the internal representation of the source code. This makes it worthwhile to perform the experiments to determine the potential for phase ordering exploration using current compiler toolchains. Although we are interested in finding suitable solutions efficiently, this does not reduce the importance of working towards having a better grasp of what factors of improvement are possible through compiler phase ordering. The results of measuring the improvements that can be achieved by compiler phase ordering specialization when compiling a representative set of kernels with a current compiler serves both as motivation for our work of developing and evaluating efficient exploration approaches, and as a baseline for comparing with the results obtained with those approaches. Notice that we do not claim these experiments will allow finding the absolute best compiler sequence for a given target platform and program, as that would require to exhaustively evaluate all possible compiler sequences. These experiments are useful to give an idea about the level of improvement that can be achieved when exploration overhead is allowed to be considerably high.

Question 2. Different target architectures can have different particularities (e.g., cache hierarchy, in/out-of-order execution) that may influence not only the existence of considerably improved solutions but also the difficulty to find those solutions. Knowing the answer to *Q2* would allow a

more informed decision when considering the use of compiler sequence specialization as a means of achieving higher optimization on a target architecture of choice. For instance, a compiler user may be better able to decide whether to use compiler sequence exploration to optimize a function/program based on data concerning the improvements previously achieved when compiling other functions/programs for the target at use, and also based on the difficulty (e.g., number of compile/evaluate iterations needed) to achieve a relevant improvement for other functions/programs regarding the metric(s) of interest. In addition, producing results of compiler sequence specialization for a number of different targets may help compiler developers identify what targets need more attention in terms of better tuning the compiler for those targets (e.g., finding better generic –Ox sequences).

Question 3. Question *Q3* asks the following of any pair of compiler sequences. If a given sequence *A* leads to higher optimization than a sequence *B* for a given function/program and target architecture pair, will sequence *A* also result in better performance than sequence *B* when compiling the same function/program to other target? Work published by other authors and our own initial experiments suggests that this is not the case for a considerable number of compiler sequence pairs. However, we thought it would be useful to perform experiments on a modern compiler targeting a number of different relevant architectures. These experiments, can not only help to strengthen this notion, but also give an idea about what potential improvement can be lost by not optimizing specifically for a given target. This is a relevant question, because if not much is lost using compiler sequences found for other targets, then not only could the compiler sequences found to be better than the traditional compiler optimization levels on a given target architecture be reused for all other target architectures (i.e., no need to repeat DSE) with considerable certainty that they would also perform better than the standard optimization levels, but also a compiler sequence (selected from a pool of compiler sequences) found to be best on a target would also rank high when compiling for others targets (i.e., when compared with the other sequences from the same pool of sequences). This would allow for instance, to optimize code by exploring compiler phase selection/ordering on a fast CPU (e.g., host CPU) when targeting a CPU/microcontroller that is difficult to integrate in a DSE loop (e.g., slow CPU, slow link for transferring code after compilation, slow or difficult to access simulator).

Question 4. A reformulation of *Q4* would be to ask what kind of characteristics programs/functions need to have in order for compiler sequence specialization approaches to work effectively. For instance, can compiler pass sequence exploration be used in all domains or is it specific of some domains, e.g., microkernels with fast execution time and/or small resource usage? Possibly, not all forms of functions/programs can in practice (i.e., because of computing power limitations or requirements to execute in a special platform) be improved with compiler sequence exploration without some modifications to the functions/programs. Modifications to a function/program (e.g., to make each DSE loop iteration faster) would have to be performed with caution, as some of them can change the sensibility to some compiler sequences. This is not a problem for a number of use

cases, as long as the final compiled version, either from the original or from the modified version, has better performance than the original version without phase selection/ordering specialization. In other situations (e.g., if the original source code went through a very rigorous validation process), it is important that the compiler sequence found to be effective to compile the modified code, is also equally suitable to compile the original code, as the latter is the one that will go to production. A methodology to adapt initially non-suitable programs/functions, even if it only works on a subset of functions/programs so that compiler sequence exploration can be performed for them would be very useful. This is the case even if it only works for functions/programs with very specific characteristics. Does an optimizing compiler phase order specially tailored for a given program with input parameters of a given shape, result in the same or similar quantitative optimization when compiling the same program with other input parameters? If the answer is affirmative, even if only for a subset of functions/programs, then a sequence found for a given kernel using a set of inputs that result in fast execution can be used for the same kernel but with larger inputs, allowing much faster exploration time in the DSE loop. If this is the case, this method could then be used to explore compiler phase selection/ordering for a function/program and target pair even in the case it would not be feasible using the original function/program input parameters.

Question 5. Question *Q5* asks what is the impact on compilation time when tuning compiler sequences for a given application and/or target using current state-of-art techniques and/or tools, how much can the overhead be reduced and how can it be accomplished. The overhead is variable and can depend on the number of compilation and simulation/execution cycles that the approach requires to find suitable compiler sequences, as well as on the compiler toolchain, the target of compilation, and the program/function to compile and its parameters. One way of reducing the exploration time is to reduce the number of simulations/executions of the compiled code/solution when relying on an iterative approach. This can be implemented for instance, by avoiding testing sequences of optimizations that are known before compilation to not improve or result in solution with the same fitness regarding the metric(s) of value (such as in (Kulkarni et al., 2010)). Additionally, the consideration of a number of static/dynamic code features (e.g., number and depth of loops) or code feature representation based on fingerprinting (e.g., see previous work in (Kulkarni et al., 2010)) can be used for pruning the design space. Other possibility is to use statistical information about the compiler sequences used in the past to somehow influence the way compiler sequences are constructed when compiling other programs (e.g., (Fursin et al., 2011)). We think it is reasonable to assume that the overhead of a system that suggests compiler sequences for a given application/target/metric tuple is a factor that compiler users will take into account to a varying degree. Compilation overhead is a metric that is important to both the typical compiler users, which tend to give higher importance to compilation time, and compiler users such as developers of mass-produced embedded systems, which may tolerate longer compilation times for the prospect of more optimized code, but also benefit from a reduction of exploration overhead by being able to explore more solution points for the same cost.

Question 6. Regarding question *Q6*, based on our experience with compiler tools, both from the day-to-day use of traditional tools such as GCC and Clang/LLVM, and the research we did in compiler tools that allow automatic exploration of compiler sequences seem to indicate that there are no other tools/toolchains capable of optimally targeting CPUs with different microarchitectures, GPUs from different vendors, and/or hardware in the form of the generation of VHDL or Verilog hardware descriptions (e.g., for implementations in FPGAs or ASICs) while relying on a unified and integrated view. We developed as part of this work, a multi-metric, multi-target, multi-compiler and multi-algorithm modular compilation flow with support for DSE of compiler phase selection/ordering. Using our system, DSE algorithms can be reused without modification for targeting different architectures, optimization metrics and using compilers. In addition, different target architectures, metrics and integration with other compiler toolchains can also be independently and pragmatically added to our compiler exploration system. This particular DSE system is an instance of a LARA (Cardoso et al., 2012a) controlled toolchain, and allows users to program DSE schemes while focusing only on the algorithm details.

Question 7. Question *Q7* asks how significantly can the overhead introduced by multiple compilations/evaluations be reduced without compromising the quality of the compiler phase selections/orderings suggested and the resultant solution (e.g., executable code) in relation to a given metric. This question asks what would be the minimum number of compilations/evaluations required by our approaches to achieve solutions comparable in quality with the ones resulting from thousands of iterations of exploration with, for instance, Genetic Algorithms (GAs) and/or Simulated Annealing (SA). This is relevant in the sense that programmers and/or other users that may be interested in optimizing their functions/programs will typically want to do it as efficiently as possible.

Question 8. A significant percentage of compiler users may not be particularly interested in embedding DSE into the compilation process because of the typical exploration overhead. This may be the case even if one is able to use exploration approaches that are able to reduce the overhead by orders of magnitude, from the often required tens of thousands of evaluations when using more exhaustive approaches to a considerably more manageable number (e.g., 100) while still being able to find solutions with comparable quality. Nevertheless, such users may be interested in DSE approaches that require evaluating only a few compiler sequences (e.g., up to 10). Question *Q8* asks how close in terms of resulting optimization could the most suitable of these compiler sequences be, when targeting a given code to a given computing device while regarding a given metric, to sequences found by iterative exploration with orders of magnitude more evaluations (e.g., SA with $10,000$ iterations) and to sequences found with other approaches that are able to reduce exploration overhead, even if not in the same order of magnitude.

1.3 Contributions

We achieved a number of goals with the work done in the context of this Ph.D. Specifically, these include publications as author or co-author about methods developed during the Ph.D. (Nobre et al., 2013b, 2014b, 2015, 2016a) and tools developed or co-developed during the Ph.D. (Cardoso et al., 2013a; Nobre, 2013; Nobre et al., 2014a), as well as studies about the compiler phase orders across different targets and different metrics (Nobre et al., 2016b, 2017). The author of this thesis co-authored the following book chapters (Nobre et al., 2013a; Cardoso et al., 2013b; Gonçalves et al., 2013) from the book "Compilation and Synthesis for Embedded Reconfigurable Systems - An Aspect-Oriented Approach".

During the Ph.D., the author of this thesis also contributed to the work presented in (Cardoso et al., 2012a; Coutinho et al., 2012b; Cardoso et al., 2012b, 2013c; Coutinho et al., 2013; Bispo et al., 2013; Cardoso et al., 2013a,c).

This Ph.D. makes the following specific contributions:

1. Novel techniques to automatically identify/suggest sequences of compiler optimizations that best suit the compilation of a given application to a given architecture;
2. Simulated Annealing (SA)-based approach with autotuning of the minimum/maximum temperature and cooling factor parameters;
3. Design space search pruning by using a graph structure computed from most frequent compiler pass sequences and relative compiler pass positioning;
4. A unified and integrated view over a compiler toolchain controlled by strategies that can be used to instruct compiler sequences and to specify DSE schemes;
5. A multi-metric, multi-target, multi-compiler and multi-algorithm modular framework for DSE in the context of compiler phase ordering, built on top of a LARA interpreter;
6. A CoSy COmpiler SYstem (CoSy) LARA weaving engine that can be attached to any CoSy-based compiler allowing fine-grain control of compiler optimizations and partial integration with the LLVM compiler;
7. Evaluation of approach that only requires testing a small set of compiler phase orders generated offline in order to achieve better solutions than achieved using the compiler standard optimization levels, experimenting with an additional parameter used during the selection of candidate sequences for this set that can improve the approach efficiency when optimizing new unseen functions/programs.

The DSE infrastructure for exploration of compiler phase orders developed in the context of the Ph.D. facilitated a collaboration with Prof. Luiz G. Martins from Federal University of Uberlândia (Minas Gerais, Brazil) which resulted in a number of publications (Martins et al., 2014b,a, 2016), and was used for the experimental work that was part of his Ph.D. thesis. The infrastructure is currently being used by Prof. Luiz G. Martins and his students.

Other contributions came from applying the tools and/or the approaches developed during the Ph.D. in different contexts. We presented results showing, for the compilation of a set of kernels from an open-access benchmark set, how energy and performance correlate when relying on compiler phase ordering specialization (see Appendix A). To the best of our knowledge we were the first to point the performance benefits of using phase ordering specialization when compiling OpenCL kernels to GPUs (see Appendix C). Still in the context of GPUs, we achieved promising results by being able to efficiently suggest compiler sequences for OpenCL kernels using a very straightforward approach using code features. Finally, as direct consequence of our work, we pass on the information about the performance improvements that can be achieved using phase ordering with Clang/LLVM, which is a widely used compiler toolchain. We believe this has the potential to attract more attention to the topic. Especially because we present improvements for a number of relevant targets/systems, including microprocessors typically found in embedded systems, which were initially our only focus, and others, including a GPU.

1.4 Structure of this thesis

This thesis is organized as follows.

The current chapter presented the motivation of the work presented in this thesis, the research questions and contributions of this thesis.

Chapter 2 introduces concepts related with compilation in general, particularly about compiler pass phase selection/ordering exploration, a more detailed explanation about the impact of such techniques, specially in the context of embedded system development. This chapter also introduces related work in the context of searching for sequences of compiler passes for hardware/software targets, in relation to the exploration algorithms that can be used, frameworks that can be used for optimization using phase selection/ordering and other orthogonal approaches.

Chapter 3 presents the compiler phase ordering exploration infrastructure proposed in this thesis. This chapter explains how the integration with the compiler backends responsible for performing optimizations and generate code is realized and how the DSE infrastructure can be used to explore compiler sequences.

Chapter 4 presents iterative approaches developed during the Ph.D. More specifically, it includes the presentation of an SA-based approach. This SA-based approach relies on a simulated annealing algorithm to focus the exploration space in a function-by-function basis, and has the useful particularity of automatically setting all SA-related variables (i.e., cooling factor, initial and final temperatures) based on the number of iterations the user of the DSE toolchain can afford. We present a graph-based approach that relies on a model created to focus the exploration space with information from previous exploration runs using other iterative algorithms, or the same algorithm in a feedback loop. We also present approaches that work in combination with the graph, such as for instance the use of the graph to guide the insertion of passes in a variant of the SA-based DSE scheme. Finally, we present an approach for suggesting a small set of compiler sequences, that relies on a special parameter to avoid overspecialization to any set of reference programs/functions.

Experimental results using these approaches are presented in Chapter 5. This chapter also presents the default experimental setup for the experiments, including the hardware/simulators and the functions/programs used.

Chapter 6 concludes this thesis and describes possible future work in the context of the work presented in this thesis.

Additional results and additional details about the results presented in this Ph.D. thesis are presented in the following appendices. Appendix A presents the results of targeting the same compiler phase orders to a supercomputer node and a single board computer, in the context of performance and energy consumption; thus presenting the relation between energy and performance (i.e., how increases in performance relate to energy savings) in two platforms from different domains. Appendix B presents experiments that explored the impact of different source-to-source transformations when targeting GPUs. Although the work presented in this appendix not strongly related with the main topic of the thesis (i.e., compiler phase selection/ordering), we still think it is related in the sense that it represents preliminary work that allowed us to realize the magnitude of impact optimization through phase selection/ordering can have when targeting GPUs. Appendices C and D present phase ordering experiments targeting a GPU and an FPGA, respectively.

Chapter 2

Background and Related Work

The use of code transformations as a way to increase the quality of code generated by a compiler, especially in the case of performance, has been extensively researched in the past decades (see, e.g., (Padua and Wolfe, 1986; Yang et al., 2000; Jiménez et al., 2002; Bondhugula et al., 2008; Renganarayanan et al., 2012)). A considerable part of that research is on static analysis to identify proprieties in the source code and optimizations (and their parameters) that can potentially improve the output code. This may include determining how a certain code construct would perform on the target hardware, and searching for code transformations that will result in improving a given set of metrics. This can be accomplished by modeling program and target platform features and/or by extensively testing a set of programs/functions on the target platform and tuning the optimization heuristics (e.g., what loop factor to use given a loop and a target platform?) and the compilation strategy to use (i.e., what passes to use and/or in what order) in each case. The drawback with the former approach, as studied in the literature, is that it tends to focus on a single or a very small set of compiler passes (e.g., tiling (Coleman and McKinley, 1995)), and therefore do not consider the impact on other compiler optimizations.

Languages, tools, frameworks and techniques have been developed by a number of authors (see, e.g., (Baxter et al., 2004; Cordy, 2004; Bravenboer et al., 2008; Yi, 2012)) to offer some automation to the process of exploring compiler sequences to specific applications, given a target platform, a compiler toolchain, and a set of requirements. The following sections separate the contributions based on the mechanism they offer to control or explore compiler optimizations.

2.1 A short introduction to compilers

Compiler design is a well established discipline, with research and development starting in the early days of computer science. The first compilers were developed in the fifties (Wilkes, 1968) and represented a milestone of paramount importance, as before computers had to be programmed using low level instructions or even using machine code directly. This effectively made programmers much more productive, as the program source code could be reused in different machines and also because it tends to be simpler to program using an high-level programming language than

relying completely on assembly instructions. We present in the next sections a number of concepts about compilers.

2.1.1 Main components of a compiler

Mapping an input program to an output program can be viewed as a process accomplished by tho phases: analysis and synthesis. These steps are respectively performed by the conceptually separable components of the compiler, the front-end and the back-end, respectively.

The front-end scans the source code, breaks it into its constituents, imposes grammatical structure, and translates a legal representation of an input program to an internal representation (IR) that the compiler back-end can operate on, through a sequence of passes including but not always limited to IR generation.

The back-end performs analysis and transformations over one or more intermediate representations (IRs) to produce an IR optimized for one or multiple metrics (e.g., performance, energy, power), and later translates it into a representation of the input program codified in one of the supported output languages. Memory dependency analysis, instruction counting, detection of single entry single exit regions, and induction variable identification are examples of operations implemented by analysis passes (Torczon and Cooper, 2011). Transformation passes include but are not limited to: evaluating expressions, propagating constant values, common sub-expression elimination, removing unreachable code, replacing operations by other equivalent but less resource intensive operations, relocating computation outside loops, tiling computation in loops to more efficiently use faster memory (e.g., cache) in the memory hierarchy, unrolling loops to allow for vectorization and reduce overhead introduced by boundary checking.

A number of authors conceptually separate generic machine code independent analysis and transformation passes from target specific transformations over the IR and code generation, by including the former in a component called the middle-end. Figure 2.1 depicts the operation of the front-end, middle-end and back-end of a compiler that generates executable code for an abstract architecture.

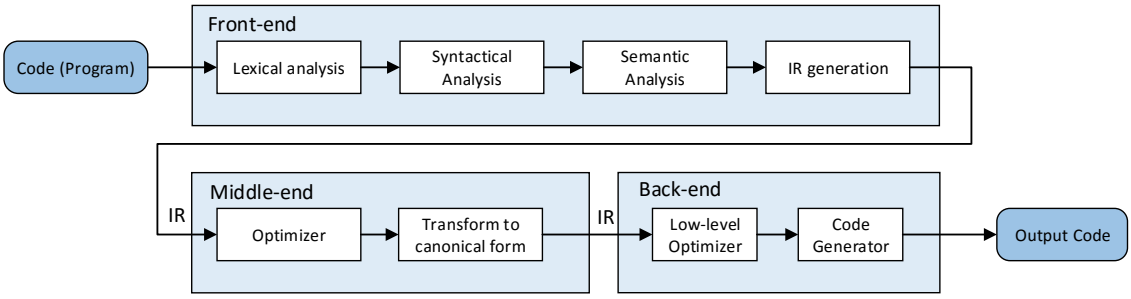

Figure 2.1: Operation of the typical components of a compiler. Adapted from (Klemm, 2008)

This separation allows for higher retargeatability, by allowing a many-to-many relationship between front-end, middle-end and back-end; as long as there is agreement on the input/output

IR format. Different input languages can be supported by only extending or developing a new front-end. An alternative or an extra optimization component can be created in the form of a new middle-end, that can replace or be executed before/after the existent one. Additional output languages, variants of both high-level and assembly languages, can be extending the existent back-end or developing new back-ends.

In case the output of the compiler is low level assembly language or machine code, then it is also the task of the back-end to manage storage of constants/variables and program instructions in memory, including deciding what variables are stored in the microprocessor registers.

2.1.2 What is expected from a compiler?

Other than the fundamental ability to generate code that is functionally equivalent to the input code, a compiler is praised if it generates code that can improve the program in respect to certain metrics when finally executed in the target platform. For instance, if a given compiler can generate executable code that is $2\times$ faster than other compiler, then one can say, without knowledge about other variables (e.g., code size), that the former is probably a better choice in scenarios where performance is important. Additionally, the more a compiler is able to generate code that complies with the non-functional requirements without too much time and resources required for the compilation process, the better. A compiler should be able to produce suitable code in a reasonable amount of time. Typically compilers take from fractions of a second to minutes to compile most programs on recent machines, with aggressive optimizations enabled. However, it is acceptable that the compilation process takes more time when considering large applications with thousands or millions of lines of code and when optimizing/generating final code for deployment.

2.1.3 How do compiler optimizations work?

A number of effective code transformations have been proposed and many have been successfully used over the past decades in research and in production-grade compilers.

Program code optimization can be accomplished by relying on compiler transformations performed at the level of the compiler IR. If correctly chosen and applied in a suitable order, they transform the IR in a way it finally results in the backend generating code that is improved in relation to the requirements of the application being compiled. Transformations to the IR are performed by compiler passes, which are the components of the compiler that perform one or more "well defined" operations on the IR, such as loop unrolling and constant propagation. Compiler passes are executed sequentially after the front-end creates the IR from the input source code and before the output code is generated. Figure 2.2 represents the application of a compiler transformation called *loop peeling* (see, e.g., (Song and Kavi, 2004)) followed by other transformation called *loop fusion* (see, e.g., (Darte, 1999; Qian et al., 2002)).

The nature of actions over the IR determines if a compiler pass is classified as an analysis pass or transformation/optimization pass. Analysis passes gather information from and can annotate the IR, while optimization passes transform the IR into a functionally equivalent IR changed in a way

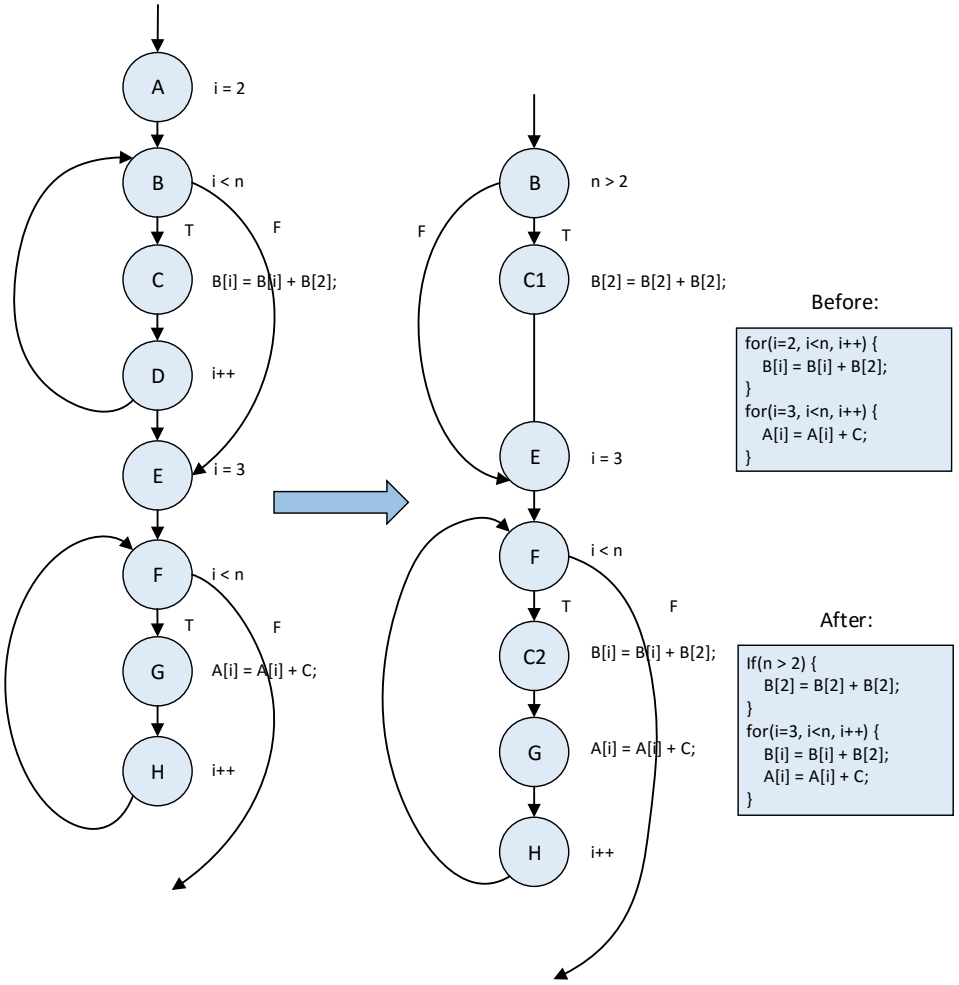

Figure 2.2: Loop peeling transformation followed by loop fusion.

that it will hopefully make the generated output code more likely to comply with the requirements. Some analysis passes can just serve the purpose of checking the validity of the IR at any given point during compilation. If the IR is found to be "broken" then the compilation terminates with an error message.

2.1.4 How can the user control which and how are compiler passes used?

Typically, compiler passes are attached to the compiler optimizer in a fixer order (e.g., GCC (Free Software Foundation, c)), deemed suitable by the compiler developers, and the user is given a chance to activate or deactivate compiler passes individually (e.g., `-funroll-loops` / `-fno-unroll-loops` activate/deactivate loop unrolling in GCC and Clang (LLVM Developer Group, a)) and/or use one of the optimization levels (e.g., GCC –O2/–O3), which represent a configuration of activation/deactivation of the compiler passes in the fixed compiler's optimization configuration. The activation/deactivation of compiler passes is referred to as *phase selection*.

Figure 2.3 depicts an example of a middle-end (see Figure 2.1) of a compiler developed using the CoSy compiler development system. In this example, the compiler middle-end has a fixed optimization pipeline, the order of execution of the compiler passes is unchangeable. The order the optimizations are executed is represented by the directed connections between the individual compiler passes in the diagram. In this compiler, it is only possible to deactivate/activate the compiler passes that are attached to the optimizer pipeline, i.e., *loopinvariant*, *loopscalar*, *dismemum*, *loopstrength*, *strength*, *looprev*, *lowerboolval* and *loopbcount*.

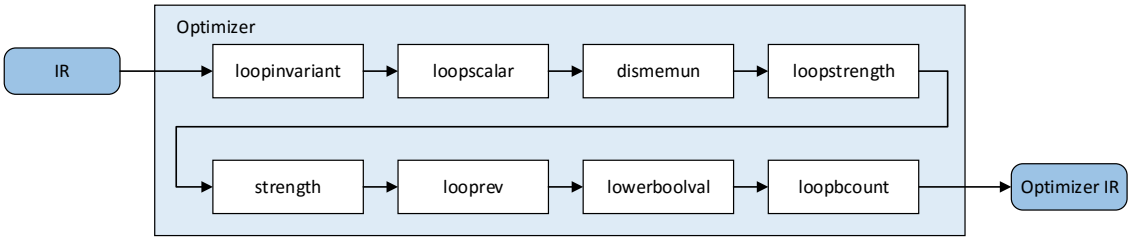

Figure 2.3: Optimizer of a compiler with a fixed optimization pipeline.

Other compiler toolchains (e.g., LLVM (LLVM Developer Group, b)) give an extra degree of freedom by additionally allowing users to specify the order in which compiler passes are executed, which is known as phase ordering. A compiler with support for phase ordering is not limited to execute the compiler passes in an order predetermined by the compiler writers.

The mechanism(s) that a given toolchain exposes control of phase ordering can, for example, be based on receiving an ordered sequence of the names of the compiler passes to execute (e.g., LLVM OPT Tool (LLVM Developer Group, b)); or based on more powerful mechanisms, such as the use of LARA aspects in the REFLECTC compiler (Nobre et al., 2013a) for selecting both the sequence of compiler passes (and their parameters) and the points of interest in the source code where the passes can be conditionally applied.

An optimizing compiler sequence is a set of analysis or transformation passes that if orderly executed by a compiler over a tool specific representation of a function and/or program, will result in the generation of a representation of the same function/program optimized regarding a given metric. The resultant representation typically assumes the form of machine code (e.g., object code or executable) for software compilation (which is the focus of the experiments presented in this Ph.D. thesis); or an application-specific architecture described in a hardware description language (e.g., Verilog, VHDL) in the case of hardware compilation.

Some compiler passes may be composite passes, i.e., execute more than one well delimited action. This has been the case since the first compilers, such as the original Fortran translator and the classic Fortran H compiler, which had support for 6 and 10 compiler passes, respectively (Cooper et al., 2002). A considerable portion of the modern compilers (e.g., GCC) follow this rule, albeit supporting many more compiler passes.

2.2 Impact of efficient phase selection and phase ordering

Phase selection and/or phase orders specialized for the input source code, non-functional requirements, and target, can lead to better software implementations. Phase selection deals with the selection of which compiler passes are executed in a given fixed compiler pipeline. A phase order is a set of analysis/optimization/lowering compiler passes executed in a given order.

The impact of an efficient compiler optimization phase order exploration system/method can be, for a number of reasons, substantial across a wide range of domains, including but not limited to the embedded systems domain and HPC.

2.2.1 Multitarget compilers may not be equally tuned for all architectures

Some compilers such as GCC and Clang/LLVM can target a considerable number of different computing devices and different instruction set architectures (ISAs). In some of the instances of those compilers, optimization may not be well tuned as in others in terms of the selection of compiler passes used and their order (if applicable) when relying on the –Ox flags (e.g., –O2, –O3 GCC flags). This can increase the potential for improvement regarding the given metrics by the use of specialized compiler phase orders.

2.2.2 Compilers are traditionally only focused on improving one or a very limited set of metrics

An additional reason for exploring alternative compiler sequences is that the generic sequences represented by the –Ox flags (e.g., –O2, –O3 GCC flags) are in most of the cases tuned to performance or code size (e.g., the –Os GCC flag), while in some situations other metrics might be more important, such as power and/or energy usage. For instance, using different code transformations, or simply changing the order of execution of the same code transformations, might result in other execution units of the processor being used, which will sacrifice performance for power/energy savings while performing the same computations.

Optimizing for performance and optimizing for energy efficiency are closely related. Total energy consumption in CMOS technology is calculated by Equation 2.1, where V_{dd} is the supply voltage, I_{leak} is the leakage current, C_{load} is capacitance and f is the operating frequency.

$$E = \int_0^t (V_{dd} I_{leak} + C_{load} V_{dd}^2 f) dt = P\Delta t \tag{2.1}$$

Static power consumption is given by $V_{dd} I_{leak}$ and dynamic power consumption by $C_{load} V_{dd}^2 f$. If power was constant during the execution of a program, faster execution would always result in higher efficiency. However, power is not constant. Specially considering current processors, which use a wide range of mechanisms in order to make computations more efficient. Power depends on frequency/voltage scaling (also known as CPU throttling), a technique for conserving power that relies on frequency/voltage pairs (also known as P-states).

Power depends on what operating/idle states (C-states, from C0 to C6 on Intel CPUs) are active at any given time. The processor alternates between an operating state (e.g., C0) and idle states (e.g., C1 to C6), that result in the internal and/or external clocks being halted, and/or the voltage being further reduced and/or the turning off the cache memory. Additionally, the activity in different processor components, such as the SIMD units (e.g., AVX2), memory caches, cores, can result in increased power consumption. Finally, other components such as the system RAM also have variable energy consumption patterns depending on their usage, which depends on a number of factors. For instance, CPU cache misses increase the energy consumption.

Because of these aspects, optimizing for performance may not be fully in line with optimizing for energy; specially when relying on fixed compiler sequences such as the ones represented by the –OX flags. Even if it was the case that energy consumption always improved in direct relation with performance (i.e., $2\times$ faster resulting in $2\times$ less energy), the use of the –OX flags (which are typically tuned for performance) would still leave potential for better performance and energy efficiency through the use of compiler sequences specially tailored for the given function and target platform pair.

2.2.3 Standard optimization levels are optimized for a limited set of functions/programs

The functions/programs used to validate the choice of the compilation sequences that are to be represented by the standard optimization levels that are distributed with a compiler, might not include any function that is representative of the functions(s) one wants to optimize. Additionally, even if functions/programs representative of the new functions(s) being compiled are used when selecting the sequences for the standard optimization levels, there is still the problem that the sequences loosing specificity by having to suit a number of different function/programs. Even if it was possible to consider all valid functions/programs when selecting the sequences to be represented by the –Ox optimization levels in a given compiler with a given set of compiler passes, it still does not seem likely that a sequence that optimally optimizes all functions/programs will exist.

2.3 Challenges

There is a number of known challenges and considerations that one should have in mind when searching for suitable compiler optimization phase orders. Some of the arguably most relevant are introduced in the following subsections.

2.3.1 Unpredictability of short and long term influences caused by particular passes

It is generally considered difficult to predict all the consequences of executing a given transformation pass. If a given compiler pass transforms the compiler IR, then the program/function representation that is accessed by subsequent passes changes. Although it is easy to accept that

certain compiler passes should be executed before other passes that rely on information produced by the former, the impact on other optimization passes in latter stages in the compilation process caused by a pass because of being on a specific position in the compilation sequence is difficult to predict with complete accuracy. This is the case even if one developer possesses deep knowledge about the compiler framework and the individual compiler passes. Optimizations are not always beneficial. For instance, executing a compiler analysis or transformation pass may alter the IR in a way that will result in the non-successful execution of other important compiler passes in latter stages of the compiler pipeline. These incidents are sometimes referred to as false interactions.

Jantz and Kulkarni (Jantz and Kulkarni, 2010) identify and study common causes of negative interactions between optimization phases and conclude that in a number of cases they are the result of false register dependence in code representation that is passed to the optimization phases. They propose ways to eliminate those false interactions with specialized implementations of register remapping and copy propagation compiler passes between other optimizations, which according to the results presented reduces the phase order search space and improves the quality of the generated code. Figure 2.4 depicts the result of applying the *instruction selection* and *common subexpression elimination* to the code depicted by Figure 2.4a, with *instruction selection* first (see Figure 2.4b) or *common subexpression elimination* first (see Figure 2.4c) applied first. The code in Figure 2.4b is better than the code in Figure 2.4c because there is one more instruction in the later, caused by false register dependence preventing the instruction selection from combining instructions 3 and 4 (Jantz and Kulkarni, 2010).

2.3.2 Challenges of selecting suitable generic compiler sequences

Compiler writers have been mostly using their intuition when determining what compiler passes are to be activated by default, as well as the order in which they are to be executed.

Compilers usually allow the user to select which optimizations to execute through the use of compiler flags. These flags result in the activation of a set of compiler passes (e.g., –Ox flags in Clang/LLVM (LLVM Developer Group, a,b) and GCC (Free Software Foundation, c)), and/or the activation/deactivation of specific compiler passes (e.g., -ftree-vectorize/-fno-tree-vectorize activates/deactivates loop and basic block vectorization in GCC (Free Software Foundation, a)). They can, in most of the compilers, only be executed in a fixed position in the pre-arranged configuration (e.g., GCC). The order of execution of compiler analysis and transformation passes is usually predetermined, configured by compiler developers to perform well on a set of benchmarks (e.g., SPEC CPU2006 (SPEC, a), MiBench (Guthaus et al., 2001), CoreMark (EEMBC)) when targeting a platform (e.g., ARM Cortex-M4 processors) or set of platforms. This is done based on the assumption that if a sequence of compiler passes performs well (e.g., results in speedups over baseline implementations) on a set of benchmarks that represents the domain(s) the compiler will target, then the same sequence is likely to generate efficient output code when compiling other applications that are not part of the same set of benchmarks. The selection of suitable program kernels to be part of the reference set of functions/programs that are used to generate the –Ox sequences can in itself a challenging task.

```
1.  r[12] = r[12] - 8;
2.  r[1] = r[12];
3.  r[1] = r[1]{2;
4.  r[12] = r[13] + .LOC;
5.  r[12] = Load[r[12] + r[1]];
```

(a) original code

```
2.  r[1] = r[12] - 8;

4.  r[12] = r[13] + .LOC;
5.  r[12] = Load [r[12] + (r[1]{2)];
```

(b) instruction selection followed by common subexpression elimination

```
1.  r[12] = r[12] - 8;

3.  r[1] = r[12]{2;
4.  r[12] = r[13] + .LOC;
5.  r[12] = Load[r[12] + r[1]];
```

(c) common subexpression elimination followed by instruction selection

```
1. r[12] = r[12] - 8;


4. r[ ] = r[13] + .LOC;
5. r[12] = Load[r[ ] + (r[12]{2)];
```

(d) register remapping removes false register dependence

Figure 2.4: Using register remapping to eliminate false register dependence. Reprinted from (Jantz and Kulkarni, 2010).

Additionally, while it is true that without using programs/functions representative of most possible code, the –Ox sequences that can be found by the compiler developers will probably not work well on many other kernels that do not share much similarities with the those programs/functions; it is also the case that the more generic the –Ox sequences are (the more possible kernels they work well with), the more potential for specialization is lost. When searching for good candidates for –Ox compiler sequences, the more the compiler developers want them to work well with a vast pool of program code, the more potential for optimizing special cases tends to be sacrificed.

2.3.3 Specialized compiler sequences vs. generic sequences

Compilers are typically distributed with a set of standard compiler optimization levels represented by flags. These flags represent fixed compiler sequences and are typically tuned for performance or code size. Compiler writers typically try to select compilation sequences that are generally suitable for the possible input codes and/or a sequence that at least is able to generate suitable code in the worst case (Chabbi et al., 2011). Those sequences are accessible using flags such as –O1/–O2/–O3 for performance, and –Os for code size.

Programmers typically rely on the standard compiler optimization levels to optimize their functions/applications. In some cases, exploring different alternative compiler sequences can lead to achieving a more desirable software/hardware implementations than the one resulting from using one of the default compiler optimization levels, such as the ones accessible through one of the default –Ox optimization flags. A number of authors have demonstrated that program- or function-specific compiler sequences tend to lead to better output code, regarding the given metric(s), than universal sequences (Cooper et al., 2002; Almagor et al., 2004; Kulkarni et al., 2010; Chen et al., 2012; Huang et al., 2015). This is potentially even more the case in compiler toolchains supporting multiple targets (e.g., LLVM), where the generic sequences are shared among multiple target architectures.

2.3.4 Problem of dimensionality of compiler sequences exploration

The exploration of compiler sequences can be a resource intensive process. Iterative approaches tend to require a number of exploration iterations in the order of hundreds or thousands in order to find compiler sequences that result in close to optimum output code regarding a given metric(s).

Moreover, the effect of individual compiler passes can depend on parameters. Compiler passes such as the ones implementing loop unrolling or other complex optimizations can have a large number of parameters, which can result in additionally increasing the exploration space by a large factor.

Equation 2.2 represents the compiler sequence space S if considering compiler phase selection and compiler phase ordering for a scenario where compiler sequences are composed of up to N

passes and compiler passes are selected (with no restriction) to compose each sequence in each n_{th} position from a list of P individual passes.

$$S = \sum_{n=0}^{N} P^n \tag{2.2}$$

For instance, if the evaluation of each compiler sequence takes on average one-tenth of a second and we are exploring sequences with up to 16 compiler passes considering a set of 32 passes to choose from (modern production compilers can have hundreds of passes), then, from Equation 2.2, it would take $\frac{1}{10}\sum_{n=0}^{16} 32^n$ seconds to explore all possible sequences. This is even optimistic as compilers typically provide tens to hundreds of compiler passes, making this a complex problem, and in some cases it may be beneficial to explore, for instance, compiler sequences with more than 32 passes. The fast growth of the sequence space even with small increases in the number of passes to consider for exploration and/or sequence length makes brute force approach impractical for most situations, thus making it imperative to use other approaches that somehow restrict the number of compiler sequences to test while still allowing to achieve suitable solutions.

Each iteration of an iterative exploration method in the context of compiler sequences consists of the generation of candidate solution(s), compilation(s), and estimation(s), simulation(s) or execution(s). The generated code can, after converted to an executable format, be executed on the target hardware, estimated, or simulated, preferably with a cycle accurate simulator. Resources (e.g., CPU cycles, memory) used by each iteration depend on the used host machine hardware/-software and the quality of the implementation and the exploration algorithm itself. Additionally, compilation time increases with increased complexity of the input source code and the used analysis and optimization compiler passes used. Execution time depends directly on the complexity of the algorithm implemented by the source code, quality of the implementation and the inputs of the application. Considering the same implementation, matrix multiplication of 1024×1024 matrices takes more time than multiplication of 64×64 matrices. The use of an estimated phase to evaluate a specific metric can substantially reduce the execution time of each Design Space Exploration (DSE) iteration with a possible loss of accuracy. In this work we do not consider an estimation stage as it is out of the scope of this thesis and can be orthogonal to the approach we propose.

The exploration of compiler sequences based on iterative approaches requires the compilation and simulation/execution of the program or parts of the program one wants to improve regarding the given objective metric(s). The simulation and/or execution of some programs/kernels can take "too long" with representative input parameters, thus posing a challenge to searching for compiler phase orders for some applications. The exploration time of a given iterative optimization phase order exploration algorithm is proportional to the time it takes for a single compilation and simulation/execution step. Approaches to reduce the simulation/execution time have been used, such as the use of a smaller but still representative set of input parameters as proposed for phase selection (see, e.g., (Chen et al., 2012)) or the use of techniques that estimate metrics (e.g., performance, energy, power) based on statistical analysis of the source code or the IR.

The objective of executing a DSE process is to find solutions that optimize a given objective

metric(s). In the cases the metric is execution time (or other directly related metric such as number of CPU cycles) of the optimized program/function, the DSE process has to be able to access a fitness value that equals or is indicative of the true execution time of the program/function. The former is obtained by executing the program and measuring its performance, while the latter can be obtained in one of a number of ways (e.g., analytical models, instruction set simulation, cycle-accurate simulation, execution on real hardware), each typically offering a different precision in the reported values. There are approaches that use predictive models to predict the execution time, or a related metric, based on information about the program/function. The program information can be, for instance, static and/or dynamic program/function features or the fitness value of other similar programs regarding the same and/or other metrics. This is an important topic because the optimized function/program has to be executed/simulated even when performance is not one of the objective metrics of the DSE, as execution and testing with pairs of known inputs/outputs is often the approach taken for verification of correctness of the optimized function/program.

2.3.5 Non-linearity of the problem space

Compiler passes can create and/or hinder opportunities for other passes. Compiler pass selections and compiler pass orders exist in a highly non-linear space, with large exploration points close to exploration points close to global minima, even when only considering a small number of optimizations (Bodin et al., 1998). Methods for finding suitable compiler selections and/or phase orders have to intelligently traverse this space, and hopefully find a local minima that is close enough to the global minima (which can be unattainable with finite exploration resources) to comply with the given non-functional requirements. It has been shown, considering small sets of compiler passes, that the optimization space has a number of local minima that are close to the global minima, but given the compiler pass selection space, the number of local minima close to the global minima will vary largely with the target architecture (Bodin et al., 1998). So not only the exploration space is highly non-linear, but it is too a high degree related with the target CPU, meaning that it may be easier to find suitable compiler pass selections for some targets, while for others a sub-optimal choice may have a considerable negative impact on the target metrics. It is therefore specially important that, for the latter case, the method used for searching compiler selections and/or orders is capable of avoiding the trap of local minima.

Although there is a large research body about the non-linear global optimization (Horst, 2002; Hansen and Walster, 2003; Hendrix and G.-Tóth, 2010), the approaches tend to focus on optimization in the context of continuous non-linear functions, while the exploration space of the compiler pass selection and ordering optimization problem is discrete.

2.3.6 Variation of optimization potential

Compiler phase order specialization does not guarantee better solutions than possible with the standard compiler optimization levels. Additionally, the improvement can depend on a number of factors; such as how optimized to the target is the source code. If it is already very optimized by

a programmer with knowledge about the target architecture and the compiler internals, then the potential for improvement with compiler optimization phase order specialization can be reduced. It also depends on the sensibility of the target to the optimizations available in the compiler and on the proximity of the solutions found with the standard phase orders to other existing improved solutions. If the factor of improvement (over compilation without optimization) when using one of the standard optimization levels is very close to the best possible, then no optimization phase order to achieve a very substantial improvement would exist. Nevertheless, in some contexts such as the embedded systems domain, even a tiny improvement is often very welcomed.

2.3.7 Effect of kernel input parameters

For some programs/kernels, the configuration of the suitable optimization phase orders may change with the input parameters. Such examples include programs/kernels with different control flow depending on input parameters, or programs/kernels that have a different performance depending on the shape of the input parameters. For instance, loop tiling may only be suggested for use when compiling a matrix multiplication kernel if the matrices given as input do not fit in cache.

The use of specialized optimization phase order found when using a set of input parameters may not result in the same improvement over the standard optimization levels of the compiler if executing the program/kernel with a different set of input parameters. In the case different parameters influence the way the application behaves (i.e., execution flow) and/or which resources of the platform where it executes are used, then it might be recommended to search for different compiler phase orders for each case. In these cases, simply using input parameters that result in faster function/program execution performance might not work as a standalone approach to accelerate DSE. In those situations, function/program-specific code modifications might be additionally required. Nevertheless, addressing research question *Q4*, we are confident that the typical functions/programs targeting embedded systems and a number of other domains (e.g., HPC) can in principle be optimized with compiler phase selection and ordering specialization.

For a simple case, such as a typical matrix multiplication kernel, the number of rows to compute for the result matrix can be reduced for DSE. Full validation would be performed only for the most suitable compiler sequences found, reducing exploration time considerably.

Other orthogonal approach to deal with functions/programs that take a long time with each individual execution, is to identify the part of code that takes the most time, extract it to a new function, and only target that part with DSE using representative parameters.

2.3.8 Assuring correctness of the generated code

Compilers must generate code that is equivalent to the source code. When generating new compiler sequences there is the possibility their use will not result in valid code, i.e., code that will generate wrong results or code that will not work at all. Eide and Regehr evaluated thirteen production-quality C compilers and, for each, were able to find cases where incorrect code to access volatile variables was generated (Eide and Regehr, 2008). When specialized compiler pass

sequences are used, there is an even higher risk of bugs in any given compiler pass not previously detected by the battery of tests performed by compiler developers. Furthermore, the longer the sequence the more likely are bugs to express themselves in compiler crashes or generation of non-functionally equivalent code.

Without going into formal verification of the optimized function/program by asserting its equivalence to a non-optimized version of the function/program, which could hypothetically be performed at the IR or assembly level, one alternative is to verity the output of the optimized program/function for a set or multiple sets of input/output pairs. The advantage of verifying it at the assembly level would be that it would work with assembly generated by different compiler versions or even different compiler toolchains, given they all generate assembly for the same instruction set architecture (ISA). The disadvantage would be that the formal verification system would have to be extended to support assembly generated for a target with a different ISA. In the case of verification at the IR level after optimization (e.g., with LLVM Opt tool), there is the advantage of being able to target a myriad of different ISAs from the same formally verified IR (e.g., ISAs supported by the LLVM static compiler). The disadvantage is that for one to thrust completely in the final optimized code this would require the backend, which receives the optimized IR as input and generates output code, to guarantee a transference of the formal verified proprieties of the optimized IR to the output code, which is not the case with compiler backends such as LLVM static compiler.

Compilers would ideally not have bugs, so one could always trust that a compiler would produce correct code. Unfortunately the compiler can be a weak link between source programs/functions that undergo through formal verification and target hardware that has been properly tested, as taking these measures does not automatically mean that correctness is transferred by the compiler to the generated code. This is specially important in the context of compilation targeting safe-critical systems where manual verification of assembly code and/or not using optimizing compiler passes is not viable. As for most use cases, the user is more likely to face problems caused by writing wrong code than it is to face compiler caused bugs, but if they do, then debugging can be very difficult (Leroy, 2011).

The problem of proving the correctness of a compiler is an old problem. McCarthy and Painter present the proof of correctness of an algorithm for compiling arithmetic expressions into machine language (Mccarthy and Painter, 1967). Their definition of correctness in the context of compilation, formalization of related concepts (i.e., formalism for description of source and output language) and methods for proof of correctness were intended by the authors to serve as groundwork for future approaches to verification of more complex compilers that serve an useful purpose. They state this challenge as “making it possible to use a computer to check proof that compilers are correct”. As of today, compiler verification has been proposed in the context of modern and undeniably useful compilers. There is an ongoing effort in introducing formal verification at the level of individual compiler passes. For instance, Vellvm (Zhao et al., 2012) is able to reason about programs/functions represented in LLVM IR and about transformations that work with that IR. Instead of finding approaches to verify the existing complex compiler toolchains, an alternative

approach consists of what is called a realistic certified compiler by (Leroy, 2009). The compiler is realistic in the sense that it compiles a relevant subset of C to the widely used PowerPC ISA, and certified because a proof assistant was used for writing the compiler and proving its correctness. The methodology used to program the compiler certifies that safety properties on the C input source code are transferred to the generated code.

Nevertheless, complete formal verification of correctness that considers the multiple combinations of compiler passes possible in the most used modern compiler toolchains such as GCC or Clang/LLVM still not a reality as of today.

2.4 Auto-tuning libraries and tuning with optimization parameter selection

Given a program/function and a target platform, there are software libraries that perform automatic code generation and/or parameter tuning to improve the program/function in terms of some metric(s). The Automatically Tuned Linear Algebra Software (ATLAS) (Whaley et al., 2000) is an example of such approach, where empirical techniques are applied in order better utilize the performance potential of the target machine, effectively making it portable across different architectures. For instance, ATLAS performs auto-tuning for a Matrix Multiplication kernel. The tuning process searches for the most suitable block sizes, i.e., the block sizes that make the data fit in each level of the memory hierarchy. Specialized cache-level blocking can result in significantly more efficient use of a given CPU.

This approach has been used a number of times. OSKI (Vuduc et al., 2005) automatically tunes sparse matrix kernels to modern cache-based superscalar machines for a particular user's sparse matrix and machine. SPIRAL (Puschel et al., 2005) automatically generates high-performance to a target architecture for linear digital signal processing kernels. UHFFT (Mirkovic, 2001), and FFTW (Frigo and Johnson, 2005) are examples of other FFT libraries targeting the CPU. MPFFT (Li et al., 2013) is an FFT Library with auto-tuning targeting OpenCL-compatible GPUs, which is specially important given the fact that the mechanism that allows achieving higher-performance and efficiency with GPUs is by exposing particular architecture features, such as the memory hierarchy.

Bodin et al. (1998) present results for compiler optimization parameter selection considering three optimizations (loop unrolling, tiling, and padding). They targeted a matrix multiplication kernel, one of the hot spots in MPEG-2, to the UltraSparc, the MIPS R10000, the Pentium Pro, and the Philips TriMedia-1000 VLIW media processor. Having as target metric the execution performance of the compiled code, they found out that the problem space is highly non-linear and that the number of local minima close to the global minima highly depends on the target processor (i.e., that is easier to optimize for some CPUs than for others). They conclude that iterative compilation is a viable approach to optimization, especially in the case of embedded systems, and achieved performance improvements between 1.8× and 10×, depending on the target. Figure 2.5

shows the interaction between loop unrolling and tiling for the Pentium II (left) and the UltraSparc (right) CPUs.

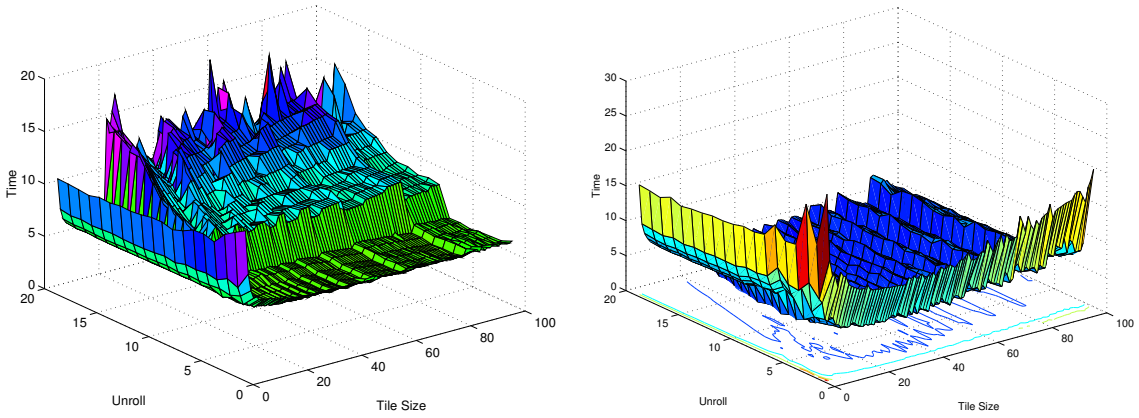

Figure 2.5: Transformation space for the Pentium II (left) and UltraSparc (right) CPUs. Reprinted from (Bodin et al., 1998).

2.5 Determining optimizations

Traditionally, compiler optimizations are applied in a fixed order (e.g., GCC –O2), predefined by compiler writers based on their experience and insight and/or based on automatic tuning for a specific, and preferably representative, set of benchmarks. In some cases, compiler optimization phase ordering can result in the generation of solutions better suited to the non-functional requirements (e.g., energy, power, performance, code size) than relying on any of the predefined compiler's optimization options. Moreover, in a number of domains, such as embedded systems, it might be more important to rely on specific code optimizations and phase orderings due to tight constraints, such as less memory/cache, a much simpler pipeline (less and simpler stages). Given the large number of compiler passes present in the most commonly used compilers (e.g., GCC (Free Software Foundation, c), LLVM (LLVM Developer Group, b)) and in other industry compilers, compiler optimization selection and phase ordering are hot-topics in compiler research. As the number of compiler passes it becomes more important to use methods that restrict the problem space to consider for exploration.

Figures 2.6 and 2.7 depict common approaches for DSE and common approaches to navigate a design space. The depicted DSE approaches are exhaustive search, a DSE scheme where all design points are tested, random sampling, which consists in randomly iterating over the design space, and knowledge-based/guided search, which consists of search methods that use information (e.g., based on fitness of design points regarding objective metrics) to focus the search for new solutions. The subsampling approaches depicted allows exploration in situations where exhaustive exploration is not feasible and/or not desirable because of a larger exploration overhead.

Random subsampling allows subsampling the exploration space in a completely unbiased way. It is used, for instance, in evolutionary methods for the mutation operations. Regular subsampling, also unbiased, consists on subsampling a regular design space, a space where the design parameters are quantized. As an example, an exploration algorithm can rely on transformation rules based on regular subsampling to determine the configuration of the next candidate solution by applying well defined transformations for the generation of the next candidate solution(s). In the context of compiler pass phase selection and/or phase ordering, an example of a transformation rule that implements regular subsampling would be a rule that replaces a compiler pass in the current sequence by a compiler pass from the list of compiler passes to explore. Subsampling sweeps consists in subsampling by performing well defined sweeps (i.e., respecting a given pattern) in a regular design space.

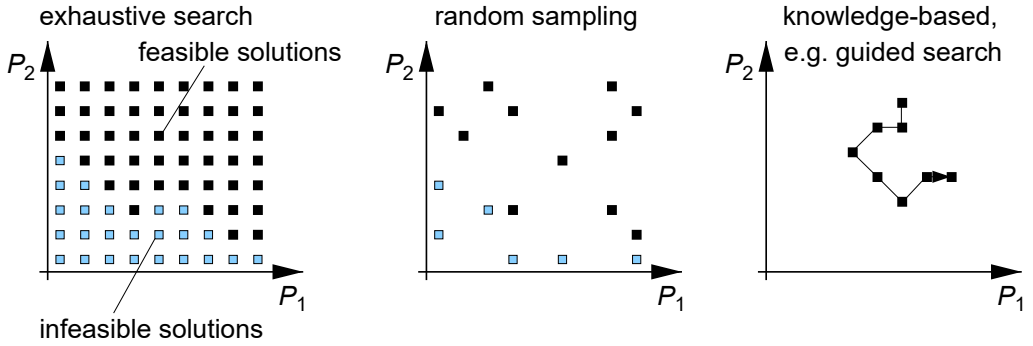

Figure 2.6: Common approaches for covering a discrete design space with two design parameters. Reprinted from (Gries, 2004).

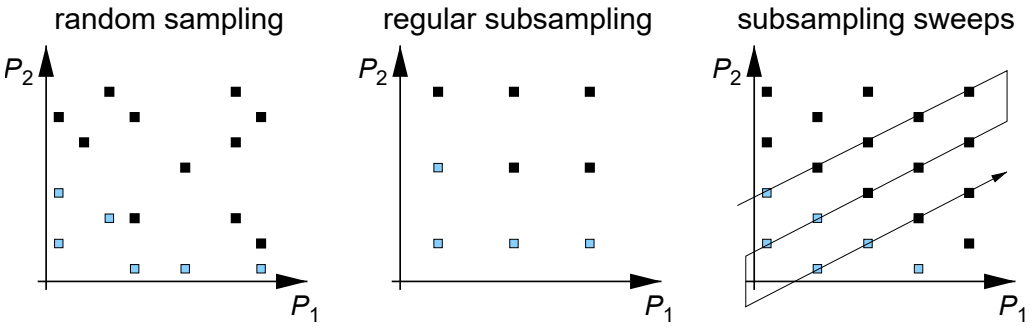

Figure 2.7: Common approaches for pruning a discrete design space with two design parameters. Reprinted from (Gries, 2004).

A number of such techniques and methods rely on heuristics for pruning the design space by reducing the number of considered compiler phase selections/orders without affecting the quality of the generated solutions. Some others rely on predictive models based on relations between static/dynamic program features, the target platform and compiler pass interdependences specific to the compiler version used. In the following subsections we present related work concerning

enumeration-based. and machine-learning based approaches. The latter rely on predictive models, for compiler optimization selection and/or phase ordering.

2.5.1 Enumeration-based approaches for phase ordering

Cooper et al. (1999) was to the best of our knowledge the first to propose iterative compilation as a means to find orderings of compiler passes that improve the quality of the compiled code with respect to a given metric. They used iterative compilation in the form of a Genetic Algorithm (GA) as a way to minimize executable footprint. The experiments to validate their approach were performed on a research compiler and considered the selection of 12 compiler passes from a pool of 10 compiler passes while targeting a Reduced Instruction Set Computer (RISC) microprocessor. Their approach was considerably more efficient than performing random probing, and was able to reduce the program size from between 19.9% and 75.5% depending on the program. They evaluated their approach with Fortran programs from SPEC95 (SPEC, c) and the Forsythe, Malcolm and Moler library (Forsythe et al., 1977), and C programs from other diverse sources. The authors noted that the reduction of executable size was accompanied with a reduction in the dynamic operation count between 20.5% and 83.1%.

Cooper et al. (2006) approach compiler optimization phase ordering evaluating different randomized search algorithms based on GAs, hill climbers and randomized sampling. Phase ordering exploration is performed at program-level and evaluated with 8 benchmarks from MediaBench (Lee et al., 1997), SPEC95 (SPEC, c) and the Forsythe, Malcolm and Moler library (Forsythe et al., 1977), targeting a simulated abstract RISC microprocessor with a research compiler, and considering 16 compiler passes/optimizations and a sequence length of 10 compiler passes. They observed properties of several of the generated subspaces of phase ordering and the consequences of those properties for the search algorithms evaluated. The randomized local search algorithms, the hill climber, and the GA-based algorithms improve the performance (measured as number of dynamic operations) of the compiled code by 15% to 30% over the compiler's generic sequence.

Almagor et al. (2004) rely on GAs, hill climbers, and greedy constructive algorithms to explore compiler phase ordering at program-level. They evaluate their approach using 10 benchmarks from SPEC95 (SPEC, c), MediaBench (Lee et al., 1997), the Forsythe, Malcolm and Moler library (Forsythe et al., 1977), and the classic sieve of Eratosthenes benchmark, to a simulated SPARC processor. They consider sequences of up to 10 compiler passes selected from a list of 16 compiler passes to consider for exploration. Using from 200 to $4,550$ compilations, the author's approach found custom sequences that achieved 15% to 25% better execution performance than the fixed sequence originally used by the compiler.

Kulkarni et al. (2004) rely on GAs to explore compiler pass sequences (of variable length) at function-level, including phase ordering exploration, targeting a simulated Intel StrongARM SA-100 processor with performance as the metric to optimize. In this work, 15 compiler passes (including loop unrolling) of the Very Portable Optimizer (Benitez and Davidson, 1988) were considered for exploration. The authors presented two approaches. One relies on avoiding unnecessary executions of the application and the other modifies the search so fewer generations are

required. The first achieved average search time reductions of 62% and the former resulted in a reduction of average GA generations by 59%, when tested with 6 MiBench benchmarks (Guthaus et al., 2001) (1 from each of the 6 MiBench categories). Additional techniques to prune the exploration space are presented by Kulkarni et al. (2009), culminating in a novel search algorithm that performs distinct compiler sequence searches simultaneously at function-level, requiring only a single program execution for evaluating the performance of potentially unique sequences for each function (Kulkarni et al., 2010). This two further approaches rely on a larger set of functions, using 12 Mibench kernels (2 from each of the 6 categories) instead of 6.

Huang et al. (2013) propose insertion-based iterative approaches for compiler optimization phase ordering in the context of hardware compilation targeting an Altera Cyclone II Field Programmable Gate Array (FPGA). Target metrics were circuit area, execution cycles, maximum operating clock frequency, and wall-clock time (extracted using ModelSim (Altera Software, 2016), an HDL simulator); and the exploration setup consisted on the LLVM-based LegUP High Level Synthesis Tool (Canis et al., 2011), considering 41 compiler passes and variable sequence length. The testbed for the algorithms consisted of 11 benchmarks from the CHStone high-level synthesis benchmark suite (Hara et al., 2008). Improvements regarding cycle latencies were between 10% and 17%, depending on the algorithm used. Algorithm 1 presents the pseudocode for the insertion based iterative approaches. The algorithm consists on individually evaluating the insertion of compiler passes (each iteration considers a distinct pass) in all positions of a candidate sequence, and accepting the configuration (i.e., the compiler sequence phase order) that results in better optimizing the target metric; if the insertion of a given compiler pass in any position results in better code. The list of compiler passes to consider for insertion can be traversed multiple times until algorithm termination; which happens when the maximum number of iterations is reached or if the list of compiler passes to consider for insertion is completely traversed without finding a better sequence than the candidate sequence.

A particularly interesting approach has been proposed by Purini and Jain (2013). They present an approach that differs from other iterative approaches in the sense that it does not rely in any of the traditional algorithms for exploration, such as GAs, hill climbers or random searches when exploring a compiler pass phase order for a given new program. Instead, the approach relies on the use of a set of compiler sequences previously found to be suitable, using iterative algorithms based on uniform random sampling and GAs, for a reference set of benchmarks representative of a number of classes of programs. Their approach circumvents program classification by relying on a small set of sequences which has the particularity of including at least an optimization sequence for each possible program class. Given a new program, each of these compiler sequences are evaluated and the one leading to better binary execution performance after compilation of the new program is selected. The approach is evaluated considering 62 machine-independent LLVM 3.0 compiler passes when generating this set of compiler sequences. Cross-validation is performed using MiBench (Guthaus et al., 2001) and Polybench 3.0 (Pouchet and Bondugula) benchmarks, resulting in an average speedup up to 14% and up to 12% for the PolyBench and MiBench programs, respectively. They found experimentally that after a number of around 10 sequences, found

Algorithm 1: *Insertion* METHOD (FROM HUANG ET AL. (2013)).

Input: Number of iterations (N), search space (S) and input function (F)
Output: Best compiler optimization sequence ($bestSeq$)

```
1  currSeq ← {}
2  currFit ← evaluate(F, currSeq)
3  refFit ← currFit
4  for i ← 1 to N do
5      bestSeq ← currSeq
6      bestFit ← currFit
7      for e ← 1 to size(S) do
8          for pos ← 0 to size(currSeq) do
9              newSeq ← putEngine(S(e), currSeq(pos + 1))
10             newFit ← evaluate(F, newSeq)
11             if newFit < bestFit then
12                 bestSeq ← newSeq
13                 bestFit ← newFit
14             end
15         end
16         if bestFit > currFit then
17             currSeq ← bestSeq
18             currFit ← bestFit
19         end
20     end
21     if currFit < refFit then
22         refFit ← currFit
23     end
24     else
25         break
26     end
27 end
28 return bestSeq
```

using their method for selecting a set of sequences that have the propriety of including at least a sequence that is suitable for each of the programs in their training set, the average performance improvement of new programs/functions, did not improve significantly. Because of the considerably small number of sequences that need to be evaluated, it is feasible in a number of cases to consider all compiler sequences in the set. This contrasts with the use of traditional iterative DSE algorithms (e.g., GAs, SA-based), which typically require evaluating a much larger number of sequences in order to find suitable compiler sequences.

2.5.2 Enumeration-based approaches for phase selection only

Chen et al. (2012) experimentally demonstrate, for compilation using GCC and Intel ICC targeting an Intel Xeon E3110 processor, that there exists at least one program-level compiler optimization phase selection that achieves at least 83% or more of the best achieved speedup across 1,000 different datasets for each of 32 programs considered (most are variations of MiBench benchmarks (Guthaus et al., 2001)). The optimal program-specific combination yields performance improvements of up to 3.75×, averaged across all datasets, over –O3 and –Ofast in GCC and ICC, respectively. These results were obtained considering 300 different compiler phase selection configurations, and considering 132 compiler options/optimizations (including loop unrolling and vectorization) when using GCC. The authors suggest adding a compiler selection phase (e.g., select between GCC or ICC), when further improvement of the performance of the compiled code is required.

Compiler optimization phase selection has been addressed in the context of Java VMs for optimization of steady-state performance of automatically detected hot-spot functions (Jantz and Kulkarni, 2013b). In this work, an iterative approach relying on GAs was used to search for optimizing phase selections, considering a set of 28 general optimizations, for a training set of functions. The resulting phase selections are correlated with relevant code features of the functions in order to build a statistical model capable of online prediction of good phase selections for unseen functions. Online prediction is possible because there is no overhead from multiple compilations such as with the iterative approaches. The experiments use the SPECJVM98 benchmark suite (SPEC, b) and 12 applications from the DaCapo benchmark (Blackburn et al., 2006), and employ cross-validation. While the phase selections found using GA resulted in average steady-state performance speedups of 6.2% and 4.3%, per-function and whole-program, respectively, the performance using the predictive model built using state-of-art statistical techniques was not able to find optimization sequences that achieve performance improvements over the default baseline, and in fact resulted in 22% worse performance than solutions found using GA and 14.7% worse than the reference baseline optimizations performed by the HotSpot JIT compiler.

2.5.3 Machine learning-based approaches

Information about the target platform and the characteristics of the function/program to compile can be leveraged to bias exploration in a way that suitable solutions can be found with less exploration overhead.

Cavazos and O'Boyle (2006) rely on a predictive model generated using liner regression to find suitable function-level compiler pass selection configurations targeting a Java VM with the Jikes RVM compiler (Alpern et al., 2000). The approach was evaluated using the SPECjvm98 (SPEC, b) and the DaCapo+ (Blackburn et al., 2006) benchmarks on a Pentium 4, resulting in an average reduction of execution time of 29% and 33% when compared with the optimization level –O2, respectively.

A GCC-based framework, named Milepost GCC (Fursin et al., 2011), is used to automatically extract program features and lean the best optimizations across programs and architectures. The framework uses a probabilistic model that correlates new program static features with the closest one seen earlier in order to suggest a custom selection of compiler optimizations. Compiler phase selection is performed program-wise and each program is classified regarding 56 features. The framework is tested considering 88 compiler passes for exploration when targeting programs from the CBench and Berkeley DB (Oracle, 2016) benchmarks to Intel and AMD x86 processors. Figure 2.8 depicts the MILEPOST open framework for tuning programs and improving optimization heuristics using machine learning.

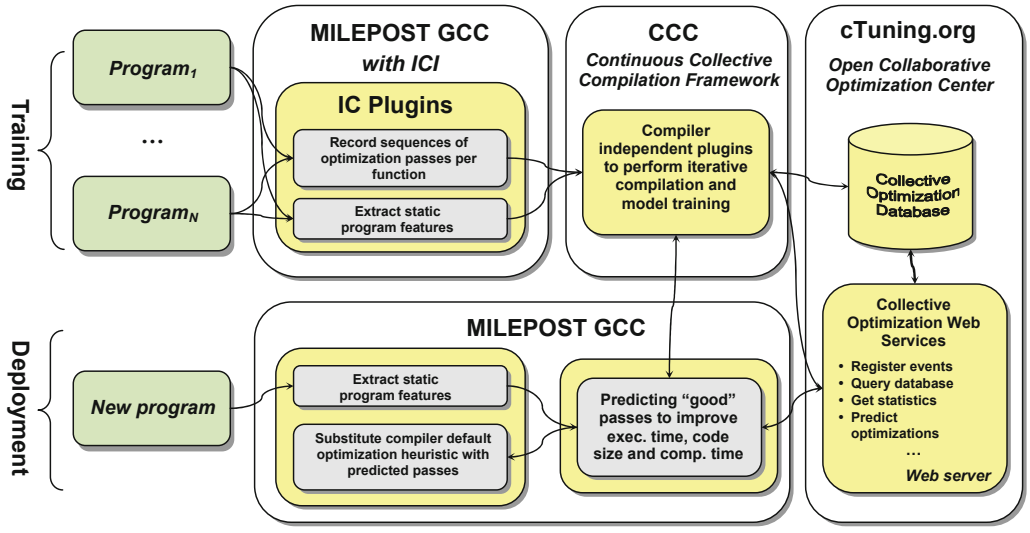

Figure 2.8: MILEPOST Framework. Reprinted from (Fursin et al., 2011).

Kulkarni and Cavazos (2012) proposed an approach that formulates the phase ordering challenge as a *Markov process* where the current state of a function being optimized conforms to the *Markov propriety* (i.e., the current state must have all the information to decide what to do next). Instead of suggesting complete compiler sequences in a single step, Kulkarni et al. use a neural network to propose what compile pass to use next based on current code features. Their approach is

evaluated by compiling hot spot methods form the SPECjvm98 (SPEC, b), SPECjvm2008 (SPEC, a) and DaCAPO (Blackburn et al., 2006) benchmarks using a Jikes RVM compiler (Alpern et al., 2000) with the neural network evolved for suggesting the compiler pass that should most likely be executed (see Figure 2.9). Kulkarni et al. reported two execution time metrics, the running time and the total time, including dynamic compilation using the neural network. The approach resulted in a geomean improvement of 10%, 6.4%, and 6.8% when considering only running time across the SPECjvm98, SPECjvm2008 and DaCapo benchmarks, respectively. If considering the overhead of extracting code features and the execution of the neural network for predicting the next pass to execute, then the geomean speedups are 3%, 4%, and 6%, respectively.

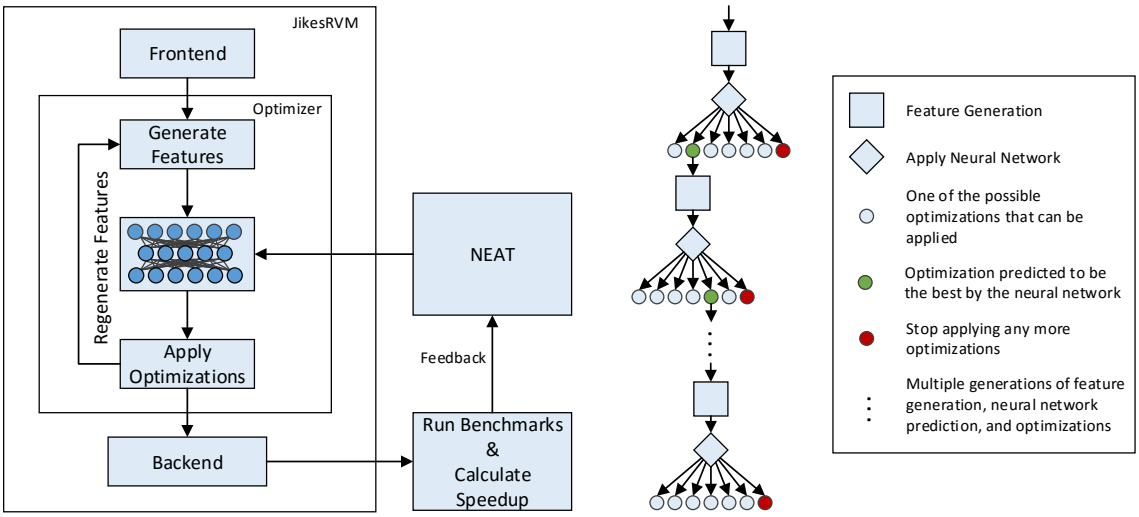

Figure 2.9: Framework used to evolve a neural network using NEAT to guide the compilation of a given method and the neural network approach for guiding the compilation process. Adapted from (Kulkarni and Cavazos, 2012).

Sanchez et al. (2011) rely on a predictive model built using SVMs to select compiler passes from an optimization level on the IBM Testarossa JIT compiler (Suganuma et al., 2000), using scalar features as input data. These features represent information about the source code, taking the form of counters (e.g., number of arguments, number of temporaries) and attributes (e.g., final?, protected?, allocates dynamic memory?, uses floating point?). The approach was able to beat IBM Testarossa in start-up performance, but not in throughput performance. The approach was evaluated using SPECjvm98 (SPEC, b) and the DaCapo (Blackburn et al., 2006) benchmarks, with the largest input available, with Oracle JVM on top of a computing platform with 16 nodes with two 2 GHz Quad-Core AMD Opteron processors each.

2.5.4 Hybrid approaches

A hybrid approach was used in (Ashouri et al., 2014) as a solution to compiler pass phase selection. A model using Bayesian networks was used to correlate machine-independent features (see, (Hoste and Eeckhout, 2007)) extracted at runtime with compiler pass selection configurations previously found for a set of training programs. Given a new program, the model generates a probability

distribution, representing different probabilities for the different considered compiler options, used to introduce a bias in the search space when sampled. Experiments were performed with 24 applications and 5 datasets from the CBench benchmark suite while targeting an ARMv7 Cortex-A9 (TI OMAP 4430) with GCC and considering 7 compiler options beyond the ones represented by the –O3 flag, including loop unrolling all loops. The approach resulted in a performance speedup of up to 2.8× (1.5× geomean) with respect to –O2 and –O3; and a 3× speedup in search time in comparison with random iterative exploration.

Agakov et al. (2006) presents a methodology to reduce the number of evaluations of the program being compiled with iterative approaches. Models are generated taking into account program features (30 features reduced to 5 using principal component analysis) and the shapes of compiler sequence spaces generated from iteratively evaluating a training set of programs. These models are used to focus iterative exploration for a new program. The authors present results concerning the evaluation of two distinct models: one based on an independent probability model and the other based on Markov chains. In their experiments they use the SUIF source-to-source compiler coupled with Code Composer and GCC, targeting programs from the UTDSP benchmark to the TI C6713 and AMD Au1500 embedded processors. The two models are evaluated with GAs in order to determine how much the design space can be pruned by the proposed approach. Cross-validation using the leave-one-out method shows that the exploration process can be accelerated by an order of magnitude, with no negative impact on performance of the generated code.

Sher et al. (2014) describe a compilation system that relies on evolutionary neural networks for program- or function-wise phase ordering exploration using LLVM. Neural networks constructed with reinforcement learning output a set of probabilities of use for each compiler pass, which is then sampled a number of times to generate compiler sequences based on the input program/function features. The DXNN (Sher, 2011) and NEAT (Stanley and Miikkulainen, 2002) neuroevolutionaty systems were used for program- and function-wise exploration, respectively. The neural networks use 48 and 44 features as input for the program- and the function-level approaches, respectively. The system was able to find compiler sequences resulting in performance improvements between 5% and 50% on a Intel Core i7, considering 53 (program-level) and 34 (function-level) LLVM compiler passes for exploration.

Martins et al. (2014b,a, 2016) propose the use of a clustering methods to reduce the exploration space in the context of compiler pass phase ordering exploration. Performing clustering on top of source code representations generated with a fingerprinting method allows the classification of a new source code into one of the existing clusters. Each cluster has associated with it only the compiler passes that are known to perform well with codes that are represented by similar fingerprints, so that the exploration space (and as direct result the exploration time) are considerably reduced when evaluating different compiler sequences with a GA-based iterative algorithm. The approach explores the use of 49 compiler passes of the REFLECTC (Nobre et al., 2013a) compiler. The GA (Goldberg, 1989) is a well-known meta-heuristic usually applied to a variety of search problems. It is an evolutionary search method which consists in generating an initial population of random solutions with a subsequent iterative evolution of their individuals, determined by their

evaluation and ranking. Each evolutionary step is called generation and is formed by simple procedures: (*i*) the selection of the parent solutions (sampled pairs of solutions); (*ii*) the application of the crossover and mutation operators; (*iii*) the evaluation of new solutions generated after the operators; and (*iv*) the reinsertion procedure which decides the survivors for the next generation. This iterative process is performed until achieving a stop criteria. In the context of compiler optimization sequence search, each solution is a sequence in the search space represented as an array of compiler passes. Experimental results reveal that the proposed clustering-based DSE approach achieved a significant reduction on the total exploration time of the search space (18× over a GA approach for DSE) at the same time important execution performance speedups (43% over the baseline) were obtained over baseline.

2.5.5 Other exploration approaches

DSE has been used in several contexts, including the exploration of hardware design and hardware/software partitioning alternatives.

Palermo et al. (2005) use a set of heuristics to reduce the overall DSE time of searching configuration of architecture of parameterizable embedded systems by up to 3 orders of magnitude, through computing an approximated Pareto set with respect to the selected figures of merit.

Palermo et al. (2009) propose an efficient DSE methodology for tuning customizable parameters in the context of designing application-specific multiprocessor systems-on-chips. The authors claim that other existing algorithms are not efficient enough for managing the application-specific constraints and for identifying the Pareto front. The methodology is able to reduce the number of simulations, by combining the design of experiments (generation of coarse view of the target design space) and response surface modeling techniques for managing system-level/application-specific constraints. This approach can be tuned to accuracy or efficiency.

Silvano et al. (2006) set-up exploration as a decision problem, relying on domain knowledge derived from the platform architecture. The number of simulations/executions is strongly reduced by only performing a simulation/execution when the impact of a given change of the parameters is not known with a high degree of certainty. They evaluated the approach with an MPEG4 encoder and an Ogg-Vorbis decoder. The parameters explored are the number of processors, two-level cache size and caching policy. An accuracy of 95% was achieved with less than 15 simulations, effectively reducing the exploration time by an order of magnitude when compared with other DSE techniques.

2.5.6 Offline vs. online compilation

Compilation can be done offline or online (also refereed to as “dynamic” compilation). Offline compilation consists in statically compiling the function/program. Typically, any subsequent execution of the function/program will use this compiled code. With online compilation (e.g., JIT compilation), the code is compiled during program execution. The latter approach can be useful, for instance, in a use case where a function/program is required to adapt at runtime to a changing

objective function. In this situation, the application can be recompiled to better adhere to the new requirements, while the current version is executing, only to be replaced when the compilation of the new version terminates. Although online compilation tends to be associated with this scenario, in some use cases offline compilation can be used to achieve the same goal through a different method. For instance, multiple versions of the same function/program can be generated by offline compilation, while still being selected for execution at runtime. An advantage over dynamic compilation is that offline compilation does not incur in compilation overhead during runtime. A possible disadvantage would be that multiple versions of the same program/function will result in multiple executables or a single larger executable including multiple optimized function variants. This can make this approach not suitable for some use-cases, where there are strict memory limitations. Nevertheless, a system for which this approach would be out of reach is also possibly not resourceful enough to dynamically compile code, but the code could be dynamically compiled by other machine. Another challenge is that it may be difficult to predict which are the suitable versions before the software and/or hardware (in the case of hardware compilation) is deployed and operational, making it difficult to successfully cover the portion of the design space that will be most suitable during execution.

As as example, Diniz and Rinard (1997) proposed the use of dynamic compilation; i.e., compiling multiple versions, and evaluating and selection one at runtime, depending on changing runtime variables, in the context of selection of suitable synchronization policies targeting object-oriented programs with a parallelizing compiler. This is accomplished with a sampling phase that measure the cost of executing each function on the current execution environment, and a production phase that executes the version found to be the most cost effective; resulting in the generated code having performance comparable with manually tuned code. As is the case of the research body related with compiler phase selection/ordering, they found that the best policy changes with the program being compiled.

2.5.7 Overview

An overview of the DSE approaches regarding iterative and non-iterative phase ordering/selection is presented in Table 2.1. The approaches can be iterative (I), non-Iterative (NI) or hybrid (H). The approach can do phase selection only (S) or phase ordering (O). The approach can work at the level of individual functions (F) or at the program level (P). Figure 2.10 presents a diagram with the related work based on the characterization of the approaches. Phase ordering and phase selection only approaches are represented in black and in gray, respectively.

A number of approaches (e.g., (Almagor et al., 2004; Kulkarni et al., 2004; Cooper et al., 2006; Kulkarni et al., 2009, 2010; Huang et al., 2013; Chen et al., 2012)) rely solely on iterative methods, while other approaches (e.g., (Agakov et al., 2006; Cavazos and O'Boyle, 2006; Fursin et al., 2011; Sanchez et al., 2011; Jantz and Kulkarni, 2013b; Martins et al., 2014b,a, 2016)) rely on predictive models to find suitable compiler pass phase orders/selections. Different from all other approaches, the approach in (Purini and Jain, 2013) iterates a considerably reduced list of compiler sequences, previously found to be suitable for a set of programs from a wide variety of

Table 2.1: Overview of automatic compiler optimization phase selection/order exploration approaches. Type: I (iterative), NI (non-iterative), H (hybrid). Kind: S (phase selection), O (phase ordering). Level: F (function), P (program).

Ref.	Type	Kind	Level	Algorithm	# Passes	Compiler	Target	Metric	Benchmarks
(Cooper et al., 1999)	I	O	P	GA	10	Research compiler	Abstract RISC machine	Code size	Forsythe, Malcolm and Moler library and SPEC + C codes
(Almagor et al., 2004)	I	O	P	GA, Hill Climbers, Greedy Constructive	16	Research compiler	SPARC	Perf.	Spec and MediaBench
(Kulkarni et al., 2004, 2009, 2010)	I	O	F	GA	15	Very Portable Optimizer	Intel SA-100	Perf.	MiBench
(Agakov et al., 2006)	H	O	F	Random search, GA + Markov models	82	SUIF + (code composer or GCC)	TI C6713 e AMD Au1500	Perf.	UTDSP
(Cavazos and O'Boyle, 2006)	NI	S	F	Logistic regression	Not specified	Jikes RVM	Intel Pentium 4	Perf.	SPECjvm98 and DaCapo+
(Cooper et al., 2006)	I	O	P	GA, Hill Climbers, random sampling	16	Research compiler	Abstract RISC machine	Perf.	Media, Spec and Forsythe, Malcolm, and Moler library
(Fursin et al., 2011)	NI	S	P	Prob. model and decision trees	88	GCC (MILEPOST)	x86 and reconf. proc.	Perf.	MiBench and Berkeley DB
(Sanchez et al., 2011)	NI	S	F	Support vector machines (SVMs)	Not specified	IBM Testarossa JiT compiler	Java VM on x86	Start-up and throughput perf.	SPECjvm98 and DaCapo
(Chen et al., 2012)	I	S	P	Random Probing	132 for GCC	GCC / Intel ICC	Xeon E3110	Perf.	Variations from MiBench
(Huang et al., 2013)	I	O	P	Sequential Insertion	41	LegUP HSL (based on LLVM)	Altera Cyclone II (FPGA)	Area, # cycles, Fmax, wall time	CHStone
(Kulkarni and Cavazos, 2012)	NI	O	F	Neural network	62	Jikes RVM	2× AMD Opteron 2216 dual core	Perf.	SPECjvm98, SPECjvm2008, DaCapo
(Purini and Jain, 2013)	I	O	P	List of 290 compiler sequences that covers all program classes	62	LLVM 3.0	Intel Xeon W35550	Perf.	MiBench and Polybench
(Jantz and Kulkarni, 2013b)	I/NI	S	F	GA / Feature-vectors	28	HotSpot Java VM server compiler	JAVA VM	Steady-state perf.	SPECJVM98 and DaCapo
(Ashouri et al., 2014)	H	S	P	Bayesian Networks	7	GCC-ARM 4.6.3	ARMv7 Cortex-A9	Perf.	cBench
(Sher et al., 2014)	H	O	Both	Neural Networks	Not specified	LLVM	Intel Core 2 Quad q8200	Perf.	bzip2 and ccbench
(Martins et al., 2014b,a, 2016)	H	O	P	Clustering + GA	49	Reflect (CoSy-based and with LARA weaver)	Xilinx MicroBlaze (RISC)	Perf.	Texas Instruments IMG and DSP programs

classes, when compiling a new program. Some approaches are concerned with the exploration of the orders in which compiler passes are executed (e.g., (Almagor et al., 2004; Kulkarni et al., 2004, 2009, 2010; Agakov et al., 2006; Cooper et al., 2006; Huang et al., 2013; Purini and Jain, 2013; Martins et al., 2014b,a, 2016; Sher et al., 2014)), while other approaches are only concerned with compiler pass selection (e.g., (Cavazos and O'Boyle, 2006; Fursin et al., 2011; Sanchez et al., 2011; Chen et al., 2012; Jantz and Kulkarni, 2013b; Ashouri et al., 2014)). Different approaches allow specialization of compiler optimization strategies to be performed at a function level (e.g., (Kulkarni et al., 2004, 2009, 2010; Agakov et al., 2006; Cavazos and O'Boyle, 2006; Sanchez et al., 2011; Jantz and Kulkarni, 2013b)), at program-level (e.g., (Almagor et al., 2004; Cooper et al., 2006; Fursin et al., 2011; Chen et al., 2012; Huang et al., 2013; Purini and Jain, 2013; Ashouri et al., 2014; Martins et al., 2014b,a, 2016)), or both (e.g., (Sher et al., 2014)). GAs are a popular choice for iterative compiler pass phase ordering/selection exploration (e.g., (Kulkarni et al., 2004, 2009, 2010; Almagor et al., 2004; Agakov et al., 2006; Cooper et al., 2006; Jantz and Kulkarni, 2013b; Purini and Jain, 2013; Martins et al., 2014b,a, 2016)). Hill climbers (e.g., (Almagor et al., 2004; Cooper et al., 2006)), random probing (e.g., (Cooper et al., 2006; Chen et al., 2012; Purini and Jain, 2013)) and greedy constructive algorithms (e.g., (Almagor et al., 2004)) are also popular choices. For the machine learning approaches, SVMs (e.g., (Sanchez et al., 2011)), decision trees (e.g., (Fursin et al., 2011)), Bayesian networks (e.g., (Ashouri et al., 2014)), logistic regression (e.g., (Cavazos and O'Boyle, 2006)), Markov models (e.g., (Agakov et al., 2006)), and neural networks (Sher et al., 2014) have been successfully used.

The number of passes considered for exploration ranges between less than ten (e.g., (Ashouri et al., 2014)) to more than a hundred (e.g., (Chen et al., 2012)). A number of different compilers have been used in the context of phase ordering or phase selection. Compilers such has GCC (e.g., (Agakov et al., 2006; Fursin et al., 2011; Chen et al., 2012; Ashouri et al., 2014)), Intel ICC (e.g., (Chen et al., 2012)), Sun/Oracle HotSpot Java VM server compiler (e.g., (Jantz and Kulkarni, 2013b)) and IBM Testarossa JiT compiler (e.g., (Sanchez et al., 2011)) have been used in the context of compiler phase selection, while other compilers with a more flexible optimizer such as the one available in the LLVM toolchain (e.g., (Huang et al., 2013; Purini and Jain, 2013; Sher et al., 2014)), the CoSy-based REFLECTC (e.g., (Martins et al., 2014b,a, 2016)) and research compilers (e.g., (Kulkarni et al., 2004, 2009, 2010; Agakov et al., 2006; Cavazos and O'Boyle, 2006; Cooper et al., 2006)) have been used in the context of pass ordering. A number of different CPUs have been the target of DSE. For instance, ARM (e.g., (Kulkarni et al., 2004, 2009, 2010; Ashouri et al., 2014)), x86 (e.g., (Fursin et al., 2011; Chen et al., 2012; Purini and Jain, 2013; Sher et al., 2014)), SPARC (e.g., (Almagor et al., 2004)), Texas Instruments DSPs and MIPS32-based (e.g., (Agakov et al., 2006)), and MicroBlaze (e.g., (Martins et al., 2014b,a, 2016)) processors have been the target of compiler pass exploration. Other targets have been, for instance, virtual machines such as Java VMs (e.g., (Cavazos and O'Boyle, 2006; Sanchez et al., 2011; Jantz and Kulkarni, 2013b)) and reconfigurable hardware such as FPGAs (e.g., (Huang et al., 2013)). The optimization metric when compiling for CPUs is usually performance, with a distinction between steady-state performance and start-performance in the context of VMs. In the case of hardware compilation (e.g.,

(Huang et al., 2013)), metrics such as circuit area and max frequency are considered relevant.

Iterative compilation is often regarded as too resource consuming and generally too expensive, but there are situations where even time consuming DSE can be very welcomed. As an example, embedded systems are often deployed in large numbers and therefore the exploration overhead that characterizes iterative compilation can arguably have a greater accumulated positive impact. Situations where the compiled code is executed multiple times, even if only deployed in one computer, can also be good candidates for iterative compilation, such as the case of scientific or industrial applications executed on supercomputers. The overhead taken by phase selection and phase ordering exploration when compiling an application can be more than enough compensated by better use of the resources available leading to faster execution and/or less energy consumption. Additionally, it is typical that programs for embedded systems and HPC are mostly limited by a set of highly localized hot spots. Although it dependents on the DSE approach, less kernels to optimize tends to result in less exploration time.

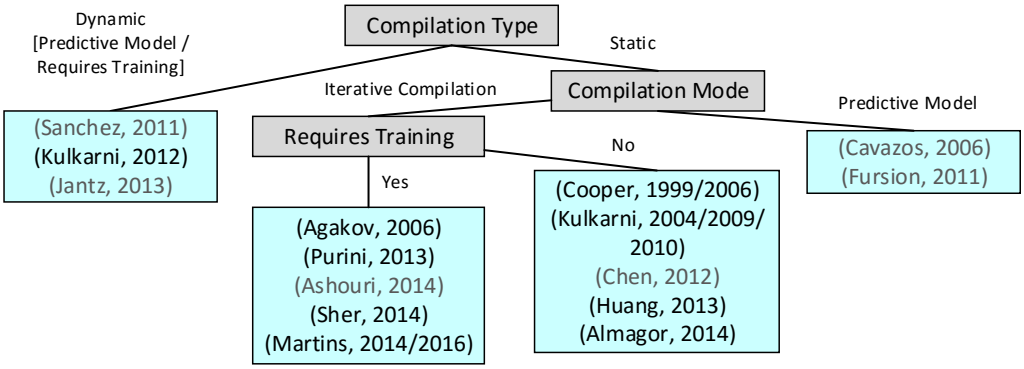

Figure 2.10: Classification of approaches in the context of the Related Work.

2.6 Programming the application of compiler optimizations

Many researchers have developed compilation systems for performance tuning, most notably in the context of scientific FORTRAN-based applications. While earlier efforts focused on empirical-based approaches (e.g., ATLAS (Whaley and Dongarra, 1998)) that relied on key output metrics to decide which sequence of transformations led to the best performing code, later efforts focused on more systematic approaches, and used a combination of performance models (e.g., (Magni et al., 2013)) and special purpose languages for defining sequences of compiler transformations (e.g., (Xiong et al., 2001)). Other approaches enabled developers to customize the composition and parameterization of transformations through scripting, to derive solutions that meet specific performance goals (Liu et al., 2009).

There have been approaches aiming at providing tool support for programming code transformations. Examples of those approaches are TXL (Cordy, 2004), ASF+SDF (van Deursen et al.,

1994), Stratego/XT (Bravenboer et al., 2008), and DMS (Baxter et al., 2004). Although they provide mechanisms to program source-to-source code transformations, they are not focused on phase selection and ordering. They, however, can be possibly used to specify compiler optimizations that can be added to the portfolio of compiler optimizations considered for phase selection and ordering. We introduce next relevant work related with tools and frameworks that were specifically proposed to assist programmers in exploring application optimizations/specializations.

2.6.1 LoopTool

Qasem et al. (2003) describe a source-to-source transformation tool for achieving higher performance without changing coding style, by providing a way to precisely control how optimizations are to be applied through code annotations on top of Fortran programs. In preliminary experiments targeting a MIPS R10K processor, the authors report the use of their tool with the Runga-Kutta advection kernel from the NCOMMAS code for mesoscale weather modeling (Wicker), the Livermore Loop 18 benchmark (McMahon, 1986), an explicit hydrodynamics kernel, and the *swim* benchmark from the SPEC CPU2000 suite (SPEC, b). They considered a set of optimizations. Statement motion, fusion, and loop alignment enable other transformations, thus they were applied to all programs. Different parameters for fusion, blocking and unroll-and-jam were explored, and some transformations were applied as composite compiler passes. Unroll-and-jam was always used with loop splitting. They evaluated each program with two problem sizes, using hardware performance counters to report CPU cycles, L1, L2, and TLB misses. Performance improvements of up to 2× were achieved on a SGI O2 workstation with a 195 MHz MIPS R10K processor by exploring different combinations of compiler optimizations (e.g., scalar replacement, storage reduction, unroll-and-jam, blocking and fusion) on the different programs.

2.6.2 X language

The X language is an annotation language proposed by Donadio et al. (2006) to specify parametrization in C/C++ programs through pragmas and macro substitution, in an effort to provide a powerful optimization framework in the context of high-performance computing. It allows application experts to automate the process of finding program parameters by representing parameterizable programs in a compact and natural way, with no required knowledge about the compiler internals. Multiple versions of the given program are generated from the programmer's annotations specifying which transformations to apply and in which part of the program, as well as their parameters, if applicable, and their order of execution. Additionally, pattern matching/rewriting rules and macro code can be used to define new transformations.

The use of the X language allows to avoid having to manually and extensively rewrite the programs in terms of the parameters for which the best values are to be experimentally determined by executing the program multiple times on the target hardware. They evaluate their approach with a DGEMM (matrix-matrix multiplication) kernel approximating the behavior of ATLAS (Whaley et al., 2000) using the X language, and claim improved productivity when compared with ad-hoc

manual C code generation. Figure 2.11 depicts a matrix-matrix multiplication kernel annotated with X language pragmas. Loops are tiled three times in order to fit blocks of data in registers and L2/L3 cache, allowing to achieve higher performance.

```
#pragma xlang name iloop
for (i = 0; i < NB; i++)
  #pragma xlang name jloop
  for (j = 0; j < NB; j++)
    #pragma xlang name kloop
    for (k = 0; k < NB; k++)
    {
      c[i][j]=c[i][j]+a[i][k]*b[k][j];
    }
#pragma xlang transform stripmine iloop NU NUloop
#pragma xlang transform stripmine jloop MU MUloop
#pragma xlang transform interchange kloop MUloop
#pragma xlang transform interchange jloop NUloop
#pragma xlang transform interchange kloop NUloop
#pragma xlang transform fullunroll NUloop
#pragma xlang transform fullunroll MUloop
#pragma xlang transform scalarize_in b in kloop
#pragma xlang transform scalarize_in a in kloop
#pragma xlang transform scalarize_in&out c in kloop
#pragma xlang transform lift kloop.loads before kloop
#pragma xlang transform lift kloop.stores after kloop
```

Figure 2.11: Matrix-matrix multiplication annotated with X language pragmas.

2.6.3 Orio

Orio (Hartono et al., 2009) is an extensible annotation-based tuning system for improving performance and productivity by allowing software developers to insert annotations in the form of structured comments into their source code to trigger a number of low-level performance optimizations. Orio generates many tuned versions of the same operation and empirically evaluates the alternatives to select the best performing version for production use. Orio has been used with an automatic parallelization tool in order to generate efficient OpenMP-based parallel code.

2.6.4 CHiLL/CUDA-CHiLL

CHiLL (Hall et al., 2010) is a unified framework that enables compilers to generate efficient code on complex loop nests, supporting a considerable number of loop transformations; including permutation, tiling, unroll-and-jam, data copying, iteration space splitting, fusion, and distribution. CHiLL can work with different application codes, by allowing flexible optimization strategies and taking into account the interactions between transformations. CHiLL can be used to program optimization strategies that generate a number of alternative optimized variants of a code segment. Transformation parameters can be adjusted and tested independently through the CHiLL script interface to code generation and empirical search. The authors reported a performance improvement

of up to 14× when comparing code transformed by using empirical evaluation with code produced by traditional native compilers. CUDA-CHiLL (Rudy et al., 2011) is a source-to-source compiler transformation and code generation framework built on top o CHiLL that allows a programmer to order the transformation of loop nests into high-performance NVIDIA GPU CUDA (Harris, 2008) code. CUDA-CHiLL can be used to express transformation sequences and code generation sequences to procude CUDA code from a high level specification. Automatically generated code was able to achieve similar or better performance than two hand-optimized GPU libraries.

2.6.5 PATUS

The PATUS (Christen et al., 2011) framework defines a domain-specific language specifically geared towards stencil computations. It is currently implemented in Java and based on the Cetus compiler infrastructure (Lee et al., 2004; Dave et al., 2009; Bae et al., 2013). It exposes a C-like syntax and a set of abstractions, such as domain, operation, and iteration, to define the core of the stencil computation over a multi-dimensional indexed grid. PATUS also allows programmers to define a compilation strategy for automated parallel code generation using both classic loop-level transformations such as loop unrolling and loop splitting, as well as exploiting architecture specific extensions such as vectorization related instructions. In the current implementation, special attention is given to cache blocking and sub-domain partitioning for these regular stencil codes. To increase the flexibility and performance portability of the generated codes, PATUS generates parameterized stencil codes, which then interface with an autotuner for specific parameter value selection.

2.6.6 POET

POET (Yi, 2012) is a interpreted scripting language for applying advanced program transformations to code in arbitrary programming languages, and to allow developing source-to-source translators between those languages in a practical way. The authors use POET to support a number of compiler optimizations (e.g., loop interchange, parallelization, blocking fusion/fission, strength reduction, scalar replacement, and SSE vectorization).

2.6.7 LARA

LARA (Cardoso et al., 2012a; Coutinho et al., 2012b; Cardoso et al., 2013a; Gonçalves et al., 2013) is an aspect-oriented domain specific language. Using LARA, developers can decompose transformations by explicitly stating the preconditions required for a specific transformation to be applied. By leveraging the join point model that LARA exposes, developers can have very detailed control of the application of code optimizations and transformations. With the composition features of LARA aspects, programmers can develop a whole range of very sophisticated fine-grain or coarse-grain strategies. In addition, LARA complements other DSE approaches by providing a unifying DSE framework (Nobre et al., 2013a), which captures and enacts evolving strategies with full design-flow control.

LARA allows separating functional and non-functional requirements. Using a LARA-based toolchain the user can develop an application while dedicating less effort to comply with non-functional requirements. These are declared LARA aspects in a separated file (use case 1), improving code mantainability. Using LARA aspects, multiple concerns/designs can be targeted/-generated with a single application source (use case 2) and multiple application sources can be used to target multiple concerns/designs using a single LARA source (use case 3). Figure 2.12 depicts these three use cases.

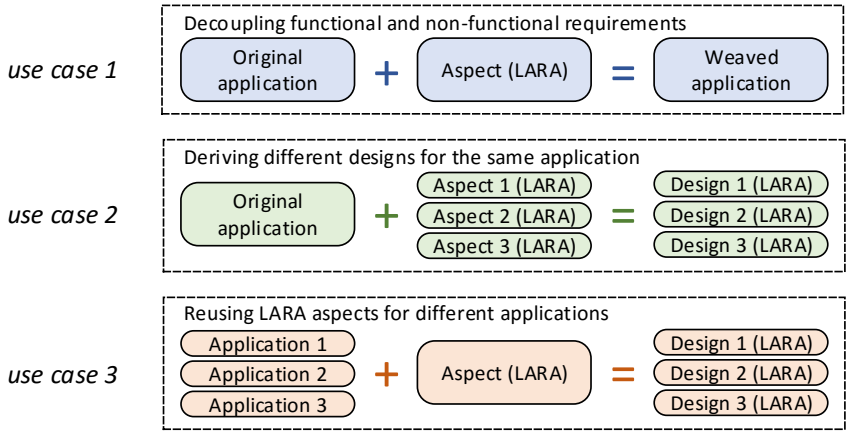

Figure 2.12: Use cases for the LARA aspect-oriented approach. Adapted from (Cardoso et al., 2013b)

Figure 2.13 depicts a simple LARA aspect that fully unrolls the innermost *for* loops of a function whose name is given as input, that are known at compile time to execute for less than 20 iterations.

```
aspectdef optimize_loops
   input functionName end
   select function{name==functionName}.loop{type=="for"} end
   apply
      optimize(kind: "loopscalar");
      optimize(kind: "loopunroll", k: "full");  // fully  unroll loop
   end
   condition $loop.is_innermost && $loop.num_iter < 20 end
end
```

Figure 2.13: LARA aspect controlling the application of loop unroll on innermost *for* loops with 20 or less iterations.

2.6.8 Overview

An overview of the tools and/or approaches introduced in the previous sections is summarized in Table 2.2. Emphasis is given to the supported input language(s), supported transformations, the

mechanism and level of control, the possibility to add extensions and the target system(s) where the tool and/or approach was tested.

Table 2.2: Overview of approaches for controlling compiler optimizations.

Framework	**Lang.**	**Transf.**	**Level**	**Input**	**Extensible?**	**System**
LoopTool (Qasem et al., 2003)	Fortran	Loop related	Optimization parameters	Loop transf. controlled by direct./pragmas	No. Tied with LoopTool loop transformation tool	Experiments on MIPS R10K (single-core)
X language (Donadio et al., 2006)	C/C++	Loop related	Optimization params.	Pragmas + Macro substitution	No.	Single-core
Orio (Hartono et al., 2009)	C	Loop related	Optimization params.	Structured C comments	Yes. Via a separated "tuning spec." file	Supports parallelization using OpenMP
CHiLL (Hall et al., 2010) CUDA-CHiLL (Rudy et al., 2011)	C/C++	Loop to CPU/GPU mapping with optimizations. Seq. of Comp. Opts.	Optimization params.	Separated transf. script	No.	Single-core and systems using CUDA-compatible GPUs
PATUS (Christen et al., 2011)	DSL	Loop related (threading, blocking, vectorization)	Parallelization and optimization parameters	Specification of the stencil operation and choice of parallelization strategies	Yes. Modular architecture of the system allows adding new components, such as back-ends	Multi- and many core processors (shared memory CPUs and NVIDIA GPUs)
POET (Yi, 2012)	C/C++ and Fortran	Loop related	Control predefined and create new optimizations	XML	No. "Loop-Processor" for transf. and an integrated search engine.	Single-core
LARA (Cardoso et al., 2012a)	C/C++ and MATLAB	Generic transformation, including loop related transformations	Optimization params., point to specific code sections, design space exploration	LARA aspect describing transformations over points of interest or design space exploration strategies	Yes. Modular architecture that allows adding new components such as compiler back ends and simulators	Target independent (e.g., PowerPC, ARM, x86, MicroBlaze)

There have been approaches for controlling specific tools using directives and constraints (see, e.g., (Qasem et al., 2003; Donadio et al., 2006; Hartono et al., 2009)). In addition to graphical

user interfaces, most solutions rely on pragma-based approaches, which pollute the code and are typically non-portable across different tools. Others have used script-based solutions such as the ones based on Tcl (Tool Command Language). While Tcl scripts can be used in a number of contexts, they neither provide advanced interfaces between arbitrary tools, nor provide sophisticated mechanisms to reference program elements of the input application code.

The Aspect-Oriented Programming (AOP) approach (Kiczales et al., 1997) addresses a number of challenges in this context, by providing mechanisms specifically to express crosscutting concerns. Not surprisingly, AOP has been intensively researched over the last decade (see, e.g., applications of AspectC++ (Spinczyk et al., 2002) and AspectJ (Gradecki and Lesiecki, 2003)). LARA (Cardoso et al., 2012a) has naturally been inspired by many of these AOP approaches. LARA has adopted mechanisms similar to the ones used by AOP approaches, but we are not aware of any AOP language with the powerful selection and composition mechanisms provided by LARA, not only regarding explicitly weaving constructs (e.g., by inserting code), but also regarding hardware/software compiler and synthesis transformations. To the best of our knowledge the LARA-based approach was the first to provide a unified and integrated view that enables developers to control and guide all tools in a toolchain, and to provide an integrated reporting mechanism supporting feedback. This feedback feature is a key differentiator of our approach as LARA aspects can capture and manipulate tool reports and use them as part of a DSE process.

The LARA framework differs substantially from the PATUS framework both in scope and expressive power. Whereas PATUS focuses exclusively on the specific notions in stencil domains and on its operators, LARA is a domain-specific language geared towards arbitrary computations. Rather than relying on specific abstractions for a given domain, LARA leverages the transformational power of third-party engines. In addition, the join point model exported by LARA allows programmers to develop very sophisticated transformation and mapping strategies clearly beyond the reach of the PATUS approach given its limited set of abstractions.

Existing high-end compilers for scientific applications (most notably FORTRAN-based) do not completely expose the set of transformations but rather only allowed a limited set of them to be controlled by developers via pragma annotations. Research efforts (e.g., CHiLL (Hall et al., 2010) and POET (Yi, 2012)) have attempted to expose source-level transformations in a controlled fashion by offering a script specification of the sequence of transformations and corresponding parameters. These analyses and transformation frameworks also differ from the LARA-based approach in various respects. They have concentrated exclusively on source code transformations for either scientific computing or, as is the case of POET, on multi-language translation focusing on the important issues of performance and auto-tuning for performance portability. Instead, the LARA-based approach bridges the gap between hardware synthesis and software compilation. Also, by directing transformation engines, LARA allows programmers to customize, in a non-intrusive way, the source code so it can be used by other tools. In addition, LARA's domain of applicability extends beyond scientific and engineering codes.

The LARA approach differs substantially from directive/annotation-based approaches such as LoopTool (Qasem et al., 2003), X language (Donadio et al., 2006) and Orio (Hartono et al., 2009).

Instead of relying on code annotations, code transformations and compiler mapping strategies are described in a separated file; allowing the reuse of this transformation/mapping specifications across different sources/applications and/or targeting multiple platforms/requirements using a single annotation-free source code. In addition, LARA offers the possibility of a finer-grain control over the transformations one wants to apply. By leveraging the join point model, developers can have very detailed control of the application of transformations, enabling development at the expression and statement levels, commonly absent from other approaches. Finally, LARA complements other DSE approaches by providing a unifying DSE framework, which captures and enacts evolving strategies with full toolchain control.

2.7 Summary

In this chapter we presented background related to compilation in general and the phase selection and phase ordering problems. We presented background related to the impact and challenges of phase selection/ordering exploration, and relevant related work in the context of DSE approaches for compiler phase selection and/or ordering.

The approaches for compiler optimization phase ordering/selection differ with respect to a number of variables. They can rely on the use of iterative and/or non-iterative methods. They can help to identify suitable compiler pass phase orderings or only compiler phase selections. Compiler optimization phase selections/orderings can be specialized at program- or function-level. They also differ in regards to the iterative algorithms used or the predictive models used, in the case of the machine learning based approaches. GAs, hill climbers, random sampling and sequential insertion based algorithms are popular choices in the case of phase ordering/selection exploration using iterative approaches. Approaches also differ in terms of the number of passes considered for exploration, the types of passes considered and regarding the exploration of the parameters of individual passes (e.g., loop unrolling factor, loop rank to unroll), and the compiler toolchain used (e.g., GCC-based, LLVM-based). Finally, they differ on the target platform and the optimization function. Metrics targeted in the context of compiler optimization phase ordering/selection are, for example: performance, code size; and circuit area and maximum clock operating frequency, in the context of hardware generation. Execution performance is the most popular metric.

The existing tools for helping with the specification of compiler transformations differ in terms of the supported input language and the target platform. Support for compiling languages such as C, C++ and Fortran is common among these tools. The target platforms can be, for instance, CPUs, GPUs or reconfigurable hardware. The type (e.g., loop related, general) and number of transformations (e.g., tens, hundreds) that can be selected and/or controlled, the control mechanism (e.g., code annotations, separated file) and the granularity of the control over optimization parameters and the points of application in the source code are examples of other variables that can differ between tools. Some of these tools also provide useful capabilities, such as the provision of a way to control external tools such as hardware debuggers/programmers, simulators/estimators and programs for visualization.

Compiler pass phase ordering/selection is a hot topic. Pure iterative phase ordering approaches may not deliver acceptable performance for practical use in most use cases outside the domain of a number of embedded systems programs/functions and/or systems that will be deployed in large scale. These approaches still introduce a considerable overhead by requiring many compilation and execution/simulation/estimation cycles, as they often have to consider tens or hundreds of compiler passes. Approaches that combine iterative search and machine learning have the potential to present a more acceptable trade-off between precision and exploration performance, than either pure iterative approaches or the pure machine-learning approaches.

Chapter 3

Guiding Compilation and the DSE Infrastructure

Design Space Exploration (DSE) in the context of compiler sequences exploration can take a considerable amount of time and resources (e.g., memory, energy, human effort), and often the toolchains/frameworks that support DSE are too tied to particular source languages, particular compilers (e.g., research compilers without open access), particular objective metrics, and/or particular target platforms.

Given the fact that this thesis is about compiler phase selection and ordering, one of our goals was to develop a software DSE system to allow us to efficiently research DSE approaches when targeting different platforms, i.e., different Central Processing Units (CPUs), or even Graphics Processing Units (GPUs) and Field Programmable Gate Array (FPGAs); and that would allow effectively targeting, with support to multiple source languages, multiple hardware/software platforms with the same DSE approaches, through tuning compilation schemes represented by compiler sequences.

Our DSE system was developed with modularity as a main priority. It allows to independently modify or add new phase selection/ordering exploration algorithms, facilitates adding support to new compiler toolchains, selecting which compiler passes to consider for exploration (given they are supported by the selected compiler toolchain), such as loop unrolling and function inlining; and modifying/specializing parameters of the existing DSE schemes. DSE schemes are programmed in LARA (Cardoso et al., 2012a; Coutinho et al., 2012b; Cardoso et al., 2013a; Gonçalves et al., 2013) and the different exploration algorithms are passed as parameters to a main DSE LARA aspect, that implements the behavior of what is common to all DSE processes. For the development of a new DSE scheme, one only has to program the new algorithm's logic, with the possibility to use other LARA-based DSE algorithm implementations as example. Newly developed DSE schemes are automatically visible to the LARA DSE environment.

The main advantage of using such DSE system is that it provides an abstraction layer, that we think, can augment productivity in the context of compiler phase selection/ordering exploration research. In fact, the effort to develop such system was largely motivated by our will to accelerate

our experiments with different DSE approaches, and facilitates targeting different targets with different compilers and possibly different metrics.

The DSE system abstracts the user from the mechanism for controlling compiler pass selection and ordering, which can differ between compilers. For instance, the REFLECTC compiler is one of the compilers supported by our DSE system. This compiler is part of the original toolchain that uses LARA to control compiler optimization strategies (Nobre et al., 2013a), and therefore compiler pass selection/ordering information is passed through a LARA aspect, which can also encapsulate information about the points of interest in the source code where compiler passes should be applied; as is mandated by the aspect oriented approach. Clang/LLVM (LLVM Developer Group, a,b), also supported by our DSE system, is one of the most popular compiler toolchains and supports the specification of compiler phase selections and orders through a command line interface using the LLVM Optimizer tool (opt) (LLVM Developer Group, e). The fact that the DSE system provides a standardized approach when developing new exploration strategies in a way that is independent of the compiler toolchain used (given that it is integrated with the DSE system), allows users to be more focused on their research instead of on particularities of the compiler used. We note however, that the kind of support given by REFLECTC is not allowed by the LLVM version used. Supporting this functionality with LLVM would require a LARA weaving mechanism.

3.1 LARA for controlling and guiding compilation

We rely on an aspect-oriented domain-specific language, called LARA (Cardoso et al., 2012a). LARA is a language mainly developed in the context of the FP7 REFLECT project (REFLECT), to describe DSE schemes in the context of compiler sequences exploration.

The first compiler to be supported by our exploration system is REFLECTC, the C to ASM/VHDL compiler of a LARA-aware CoSy(ACE)-based toolchain we developed in the context of the REFLECT project. Our DSE system also supports exploration/compilation using Clang/LLVM (LLVM Developer Group, b). LLVM-based compiler sequences exploration setup allows to target a number of relevant computing devices/platforms, as there are LLVM backends for a myriad of architectures/platforms. We think support for LLVM is important, as it allows experimenting with a myriad of architectures/platforms; such as subsets of the x86, ARM, LEON3 microprocessor architectures and GPUs. Our DSE system also supports GCC, but given the fact that it only supports phase selection (i.e., it does not support phase ordering), we favored the use of LLVM for the experimental work related with this thesis.

Support for compilers (i.e., REFLECTC, LLVM, and GCC) is integrated into the DSE system in a way that it does not require the need to maintain alternative compiler sequence DSE systems for the different compilers. In addition, our system is organized in a way that facilitates adding support to other compilers with minimal implementation effort, without requiring modifications to the DSE system's code that supports the use of the currently supported compiler toolchains.

The DSE system is capable of supporting different compilers across different DSE algorithms and different optimization metrics. Most experiments presented in this thesis are performed using this new unified exploration system. Our system is able to compile the same set of benchmarks for a number of different targets, while using a single specification of each of the DSE algorithms, independently of the compiler and objective metric (e.g., performance, energy use, power, executable size).

Using LARA allows us to write DSE algorithms while abstracting ourselves from the tools and technologies used by the LARA-aware toolchain in use, as LARA provides a mechanism for querying information and/or applying transformations/optimizations to specific parts of source code, using "select/apply/condition" code sections and other code using a JavaScript-based syntax. The LARA aspect-oriented compilation model is implemented by a LARA interpreter, implemented by a tool called *larai* (Cardoso et al., 2013b), which is capable of executing LARA outer-loops; and a set of LARA weavers that have access to the LARA-aware compilers/tools that are part of the used toolchain. The *larai* interpreter internally uses *larac* (Cardoso et al., 2013b), a LARA to XML representation compiler. The LARA outer loop mechanism is used to implement the execution of DSE schemes. The REFLECTC CoSy-based compiler is an example of a LARA-aware compiler that relies on a LARA weaver implemented as a compiler engine that reads LARA aspects feed to REFLECTC and dynamically configures the compiler pipeline in a way it reflects what is expressed by the aspects given as input.

It is important to notice that a tool used on a loop controlled by *larai* does not need to be LARA-aware. For instance, in our context of DSE of compiler sequences using our DSE system with the LLVM toolchain or GCC, we did not yet augment the compiler to be internally LARA-controllable. In the case of LLVM, we select which optimizations we want to execute and in which order by generating an ordered list of command line flags and passing them to the LLVM Optimizer tool when called from the LARA execution instance of the DSE process. Only if one needs to have more control over LLVM or GCC, such as we currently have with REFELCTC, would it be required to augment the compiler to accept LARA aspects as input, changing the way LLVM or GCC compiles in order to reflect what the aspects represent (i.e., conditional actions over points of interest in the source code). A number of LARA's capabilities, which are at this time only implemented in REFLECTC, such as the possibility to query information about a given input program and to apply actions of specific points of interest of the program were not generally used in the context of our compiler sequence exploration experiments. Nevertheless, we give a brief introduction of the said capabilities, implemented as a weaving engine/pass in REFLECTC, as we used them in the experiments targeting C to VHDL (Nobre, 2013) to explore optimization parameters accessible in REFLECTC through this fine-grain control (e.g., loop ranks and loop unrolling factors for the loop unrolling compiler pass) in addition to exploring the order in which compiler passes are executed. Additionally, these capabilities can be used in future for, e.g., exploring compiler sequences at other levels (e.g., loop-level) other than at the function-level and/or only executing certain compiler passes if a set of given conditions described in LARA are met, using REFLECTC or using LLVM with a LARA weaving engine developed in future.

3.2 Controlling compilation in CoSy

CoSy (ACE) is a commercial, easy targetable (i.e., has a number of code generators, and allows to create new code generators) and highly flexible (user can create new analysis/transformation engines), compiler development system developed and with support by ACE – Associated Compiler Experts bv. It is used to develop high-quality and high-performance compilers for a broad spectrum of target systems, ranging from 8-bit microcontrollers to Complex Instruction Set Computer (CISC), Reduced Instruction Set Computer (RISC), DSP, and 256-bit Very Long Instruction Word (VLIW) processor architectures.

Given its highly modular design based on an extensible IR and the available code generators, the CoSy environment enables the construction of production-quality performance compilers in a highly efficient manner. CoSy allows a programmer to create specialized compiler instances, using ACE-provided compiler engines and user created engines in a flexible pipeline. Engines are developed in C/C++ with complete access to the IR, called CoSy Common Medium Intermediate Representation (CCMIR) (ACE, 2008).

3.2.1 The REFLECTC CoSy compiler

We developed a CoSy-based compiler, called REFLECTC (Nobre et al., 2013a), in the context of the REFLECT project (REFLECT). The REFLECT toolchain targets hardware/software platforms. The REFLECTC compiler targets the X86, PowerPC and MicroBlaze (a Xilinx softcore) ISAs, and is capable of generating hardware description files in the form of VHDL code for portions of code (e.g., functions) marked by the user for C-to-gates compilation. The VHDL CoSy backend engine used by the REFLECTC compiler is a CoSy implementation of the DWARV (Nane et al., 2012) high-level synthesis tool from TU Delft. An important feature of REFLECTC is that it can optionally be controlled/guided by LARA. Particular optimizations, and their order in the compiler pipeline, are controlled with a description of compilation steps programmed in LARA.

3.2.2 Control/Data flow graph and aspect intermediate representations

The conceptual IR used in the REFLECT design-flow consists on a tuple (CDFG-IR, Aspect-IR), which represents two distinct IRs that capture side-by-side information across the design-flow:

- Control/Data Flow Graph (CDFG-IR): captures the computation elements of the application. Internally, this IR is annotated with information that aids a compilation and synthesis tool in the mapping of the input computations to the target platform. The CDFG-IR allows the representation of data-structures and computations while maintaining a consistent representation over the design-flow, from the application to the VHDL-RTL generation in the case of reconfigurable hardware, or assembly code in the case of a target processor.

- Aspect-IR: this IR captures the flow of information obtained from the LARA aspect-oriented specifications using the LARA front-end.

The LARA-IR is thus a combination of a CDFG representation and an aspect specification. As these two types of representation denote very distinct concerns they are naturally decoupled. The CDFG captures the intrinsic algorithmic functionalities of the representation – what needs to be computed. The Aspect-IR defines how that computation can be carried out and where, if possible, and under which mapping constraints.

3.2.3 LARA weaving in CoSy

Within CoSy, the CDFG-IR corresponds to ACE's CoSy intermediate representation – CCMIR. Therefore, CCMIR is the vehicle in which the weaving process operates at the CDFG-IR level in the REFLECT design-flow. As CoSy allows the execution of a sequence of optimization engines, a simple approach for controlling the various optimizations led to the development of a mechanism that includes information pertinent to each optimization, but still allows it to be scalable without compromising the existing infrastructure. To this extent, we rely on user-provided aspect specifications, which are converted to Aspect-IR (in an XML representation) by the LARA front-end. Each weaver uses the Aspect-IR to weave representations in different stages of the compilation flow with the aspect definitions. This mechanism is partially outlined in Figure 3.1, in which the combination of components takes effect during the weaving stages. A specific set of abstractions called join points identify the structures in the CDFG-IR to which specific advices in the Aspect-IR description are applied. The join points thus allow programmers to bridge the gap between these two representations in LARA-IR.

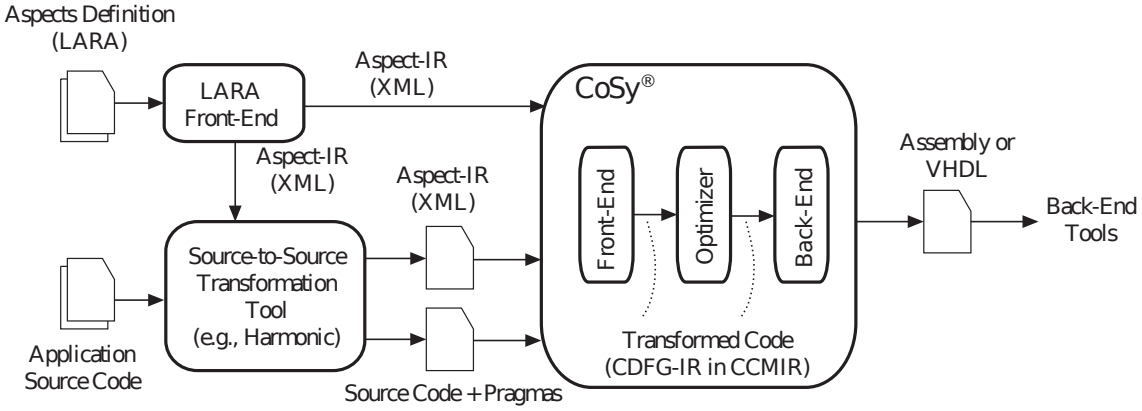

Figure 3.1: Use of aspects to guide and control source-to-source transformations and CoSy engine optimizations. Reprinted from (Nobre et al., 2013a).

Although not used in the context of this thesis, an extended version of Harmonic (Luk et al., 2009), the source-to-source front-end tool used in the REFLECT's design-flow, is responsible for transforming input C code according to LARA strategies and is mainly responsible for code insertions for application monitoring and for hardware/software partitioning. This first process is followed by the front-end engine of CoSy that transforms the input C program and annotations to

the CDFG-IR. Within this framework, a programmer specifies a set of aspects and pointcut definitions for the application to which the weavers apply a set of strategies. The weaving processes produce different types of results according to the REFLECT design-flow stage and to the content of each executed aspect. The CDFG-IR represented using CoSy's CCMIR captures not only the behavior of the application, but also a low-level representation of the computations. The Aspect-IR (Coutinho et al., 2012a) contains program locations (join points), the corresponding actions which are applied on selected join points, and the conditions in which they are applied. Each weaver operates independently and only based on its competency, ignoring other aspects outside its scope. For example, an aspect can specify the instrumentation of a particular loop and its partial unrolling. The loop is thus subject to two actions performed by two weavers, e.g., Harmonic does the instrumentation, while REFLECTC performs partial unrolling. Both of them are driven by the same Aspect-IR description. The CoSy weaver engine, named *cweaver*, was co-developed within the context of the REFLECT project and is responsible for executing LARA aspects as described by their IR (Aspect-IR). Furthermore, this engine forces the execution of other CoSy engines according to the optimization actions described in these aspects. The CoSy weaver engine, thus, allows developers to control and explore sequences of compiler engines from more than a hundred engines provided in the CoSy distribution. In addition, this weaving engine can be used to develop CoSy compilers that can process LARA compilation strategies and target different platforms, including processors and reconfigurable hardware. In REFLECT, the CoSy weaver engine is integrated in the REFLECTC compiler. The CoSy weaving process is responsible for querying attribute values from the CCMIR, get pointers to CCMIR nodes representing join points, and to invoke optimization engines in a specific order based on aspect descriptions. The interaction between the CoSy weaver engine (cweaver) and other CoSy engines around the CCMIR structure is guided by the Aspect-IR description. The example in Figure 3.2 illustrates the REFLECTC flow for generating x86 assembly code. A number of other code generators (e.g., ARM, x86, PowerPC, MicroBlaze) can be used instead.

Figure 3.3 depicts a simple LARA aspect that performs a sequence of optimizations on for-type loops of a function whose name is given as input. Furthermore, the optimizations are only applied to innermost loops with less than 20 iterations. Variables *functionName* and *factor*, input parameters of the aspect, represent the name of the function targeted by the aspect and the unrolling factor for the loops considered for optimization, respectively. Optimizations are applied in the order they appear in the apply section of the aspect. The values for the attributes *is_innesmost* and *num_iter* used in the condition section of the aspect are statically determined by the compiler. For the for-type loops for which the compiler cannot provide the number of iterations (given by the *num_iter* attribute) the *apply* is not executed.

Internally, the *cweaver* engine communicates with the LARA interpreter using a remote procedure call (RPC) scheme. Both components communicate and synchronize using a shared data structure. The LARA interpreter executes the Aspect-IR and for each attribute used or CoSy required action, it invokes the core *cweaver* and waits until *cweaver* flags that a particular weaving operation on the CCMIR is completed. Operations include performing an optimization, extracting

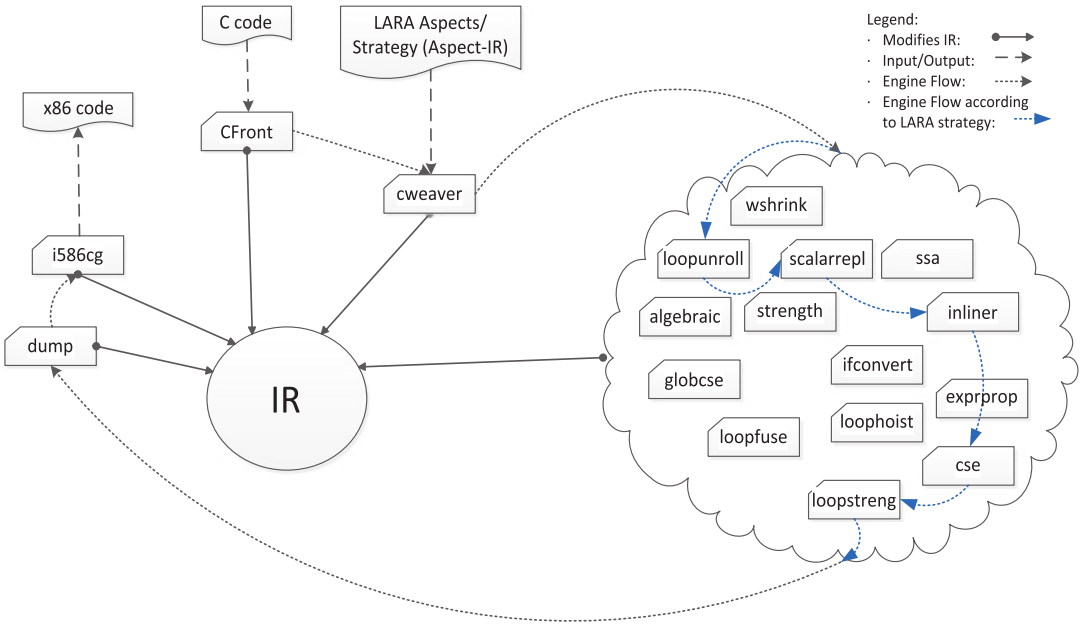

Figure 3.2: Weaver interaction within the CoSy framework for generating x86 code as output. The i586cg includes the default sequence of engines to target x86. Reprinted from (Nobre et al., 2013a).

```
aspectdef  optimize
  input functionName, factor end
  select
    function{name==functionName}.loop{type=="for"}
  end
  apply
    optimize(kind: "loopinvariant");
    optimize(kind: "loopstrength");
    optimize(kind: "loopscalar");
    optimize(kind: "loopguard");
    optimize(kind: "loopunroll", k: factor);
  end
  condition
    $loop.is_innermost && $loop.num_iter<=20
  end
end
```

Figure 3.3: LARA aspect controlling the application of a sequence of optimization on innermost for-type loops with 20 or less iterations.

an attribute from the CCMIR, or getting CCMIR nodes related to join points.

The communication between the LARA interpreter and the core CoSy *cweaver* engine is performed using three weaving commands: *jpq* (join point query), *attq* (attribute query), and *act* (ask the execution of an action, typically through the invocation of an engine). The LARA interpreter (triggered by the input Aspect-IR description) supplies the CoSy *cweaver* with a sequence of weaving commands to be executed within the CoSy environment. The CoSy cweaver is thus responsible for computing join points (in essence CCMIR nodes), extract attribute values from specific CCMIR nodes, and executes engines on target CCMIR nodes. Table 3.1 presents examples of join points and attributes supported by the current weaving process. The join points supported include functions, loops and variables; and each join point has a specific set of attributes.

Table 3.1: Examples of attributes extracted by the CoSy *cweaver* engine

Join point	Attributes	Return	Description
loop	is_innermost	{1, 0}	Loop is innermost?
	is_outermost	{1, 0}	Loop is outermost?
	numiterisconstant	{1, 0}	Number of iterations is constant?
	numiter	int	Number of iterations?
	type	{“for”, “do while”, ”while”, “unknown”}	Type of the loop
	rank	1[0-9]*(:1[0-9]*)*	Identification of the loop in nested loops or in sequences of loops in a function
	haschanged	{1, 0}	CCMIR of the loop has changed?
	numloops	int	Number of loops in the function
	numExprs	int	Number of CCMIR nodes of type expression related to the loop
function	name	String	Name of the function
	hascall	{1, 0}	Function includes calls to other function?
	numargs	int	No. of parameters of the function
	usespointers	{1, 0}	Function uses pointers?
	usesarrays	{1, 0}	Function uses arrays?
	numExprs	int	No. of CCMIR expression nodes in function
	haschanged	{1, 0}	CCMIR of the function has changed?
	numloops	int	No. of loops in the function
	usesdoubles	{1, 0}	Uses variables of type double?
	usesfloats	{1, 0}	Uses variables of type float?

The CoSy weaver uses the same semantics of the generic LARA weaving. Figure 3.4 shows the actions performed by the weaver when considering the list of join points that resulted from the corresponding *select* section. The LARA code of the corresponding *apply* section is executed for each join point in the list if the *condition* section is evaluated to true. One important mechanism of the weaving process is the extraction of the attributes used in the *condition* and *apply* sections. The sequence of actions shown in Figure 3.4 is repeated for each *apply* action in the LARA aspect being executed.

```
Input: CCMIR, Apply Section and its JoinPointList
Output: Modified CCMIR according to optimizations
begin
  foreach JPoint in JoinPointList do
     Evaluate Condition (if exists) of the Apply section related to the Select(s)
     section(s) responsible for the pointcut expression, including:
     [extract attribute values used in condition from CCMIR according to JPoint]
     [if condition evaluates false then continue foreach]
     Execute Code of the Apply section related to the Select(s) section(s)
      responsible for the pointcut expression, including:
       [extract attribute values used in condition from CCMIR according to JPoint]
       [execute LARA optimize instructions:
      Instruct compiler pipeline to execute CoSy engine (with parameters)]
  end foreach
end
```

Figure 3.4: Execution of the CoSy weaver (cweaver) steps related to LARA apply sections. Reprinted from (Nobre et al., 2013a).

3.3 Controlling compilation in LLVM

LLVM (LLVM Developer Group, b) is a compiler infrastructure written in C++ for performing static/compile-time optimizations on an IR generated from source code written in one of multiple supported source code languages, including C/C++, Objective-C, Fortran, Ada, Haskell, Java, Python, Ruby, ActionScript, and GLSL. The compatibility with C, C++, Objective C and Objective C++ as source languages is provided by the Clang front-end (LLVM Developer Group, a). Alternatively, *llvm-gcc* (LLVM Developer Group, c) (using the `-emit-llvm` flag) can be used as C front-end for the LLVM compiler, with support to many of GCC's features, options and C language extensions.

The use of LLVM with the Clang frontend, or other frontend for one of the non C-like programming languages, is increasingly accepted both in the academic environment and in industry. Despite LLVM being quite new when compared with GCC, the fact it is built to be modular and reusable resulted in the existence of an impressive and rapidly increasing number of compiler toolchains using it as middle-end (i.e., for analysis and transformation/optimization) and backend (i.e., code generation, usually assembly for a given architecture). New analysis and transformation compiler passes can be developed using C++ and added to LLVM. New code generators for any

microprocessor ISA or other computation device can also be developed and added to the compiler. LLVM's source code is available under the University of Illinois/NCSA Open Source License.

Current versions of LLVM are able to generate code for a number of targets (supported targets may change with future versions), including but not limited to, ARM, Hexagon, MicroBlaze (up to version 3.3), MIPS, NVIDIA PTX, PowerPC 32/64-bit, SPARC (V9), System z, X86 32/64-bit, and XCore. Additional code generators are part of the official LLVM build as experimental backends, and many others exist outside of the main LLVM build. LegUp (Canis et al., 2013) is a notable example of an alternative LLVM toolchain. LegUp is capable of partitioning, individually optimizing and generating code for HW/SW systems based on a MIPS (Tiger) softcore and HW accelerators for functions specified by a programmer/user. Other alternative LLVM toolchains include Nios2-LLVM, for Altera Nios II embedded processor softcore; and LLVM-Tilera for Tilera accelerator processors, and a number of others.

3.3.1 The LLVM optimizer tool

The selection of which compiler passes to use and their execution order with LLVM is straightforward, as the LLVM toolchain allows the selection of what compiler passes to execute and in which order by passing their names in an ordered list by command line to the LLVM Optimizer tool (opt). The LLVM Optimizer tool takes as input a representation of a program in LLVM IR generated by the Clang C frontend. All the programs we use as benchmarks for our compiler sequence exploration approaches are represented in the C programming language. After the LLVM Optimizer applies the transformations in the order represented by the compiler sequence passed as input, the DSE system calls the LLVM static compiler (llc), that is used to generate code for a number of target architectures. We have developed an interface between the LARA framework and LLVM, which allows calling the Clang front-end, the LLVM Optimizer and the LLVM static compiler from LARA aspects. Relying on a simple interface, a LARA programmer can access the Clang/LLVM toolchain and instruct the compilation of any given source code considering the execution of a sequence of compiler passes at an arbitrary order.

3.3.2 LLVM with the standard optimization levels

The LLVM optimizer tool supports the typical -Ox flags that represent standard optimization levels. The result of using one of the standard optimization levels is the execution of a number of compiler passes in a given order, decided by the compiler writers.

The passes and the order of execution in the compiler phase orders represented by the –Ox flags have been meticulously tuned to C/C++ by the compiler writers (LLVM Developer Group, f). As an example, in LLVM 3.9 the *opt* –O3 sequence includes the execution of a total of 172 instances of compiler passes, from which only 72 represent the execution of distinct passes. For reference, the *opt* –O2 sequence is the same as *opt* –O3, except for not having the *argpromotion* pass, making it 171 passes wide (71 distinct passes). The *opt* –O1 sequence is smaller, with 159 pass instances, of which 65 are distinct passes). Compared with –O2, it does not apply *constmerge*,

elim-avail-extern, *globaldce*, *gvn*, *inline*, *mldst-motion* and *slp-vectorizer*, but adds *always-inline* after *alignment-from-assumptions* (at the fourth position in the sequence).

3.4 DSE loop infrastructure

Our DSE schemes are implemented by writing LARA aspects that rely on an outer loop which controls calls to the toolchain tools. This outer loop can choose and send parameters to different tools, and uses a feedback mechanism to read information from tool reports in an integrated way. A DSE scheme is implemented with a LARA aspect implementing the DSE algorithm. This LARA aspect implements an outer-loop and is called by the main DSE system LARA aspect, which is passed to the LARA interpreter tool (*larai*) (see Figure 3.5). Other tools (i.e., compilers, simulators, interfaces to hardware) controlled by this LARA outer loop can also receive LARA aspects as input for a fine-grained control of the compilation process, such as optimization parameters and points of application of specific optimizations.

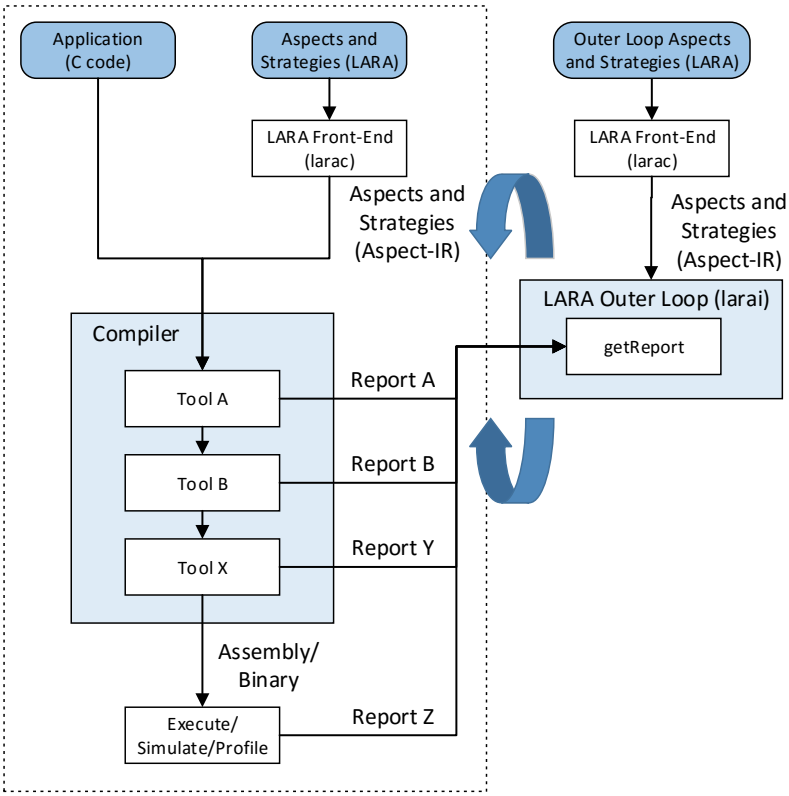

Figure 3.5: LARA based toolchain flow and the LARA outer loop mechanism when using REFLECTC. Adapted from (Cardoso et al., 2013b).

Figure 3.6 depicts a code example showing how an iterative DSE algorithm can be implemented in LARA, relying in the LARA outerloop mechanism.

Generically, a LARA DSE algorithm implementation receives as input a list of phases to select/order during exploration, the maximum number of DSE iterations to perform and the maximum

```
import DseRun;
aspectdef DseAlgo
        input exporeParams = [], maxIterations, maxWidth, additionalParams; end

  outerloop:
    for(var i=0; i < maxIterations; i++) {
        ...
        // 1. Generate new phase order/selection configuration(s) with 'maxWith'
            maximum number of passes
        // 2. Compile and evaluate using new phase order/selection configuration(s)
             by calls to 'DseRun.compile_execute_and_report(compileParams)', which
            returns a fitness value
        // 3. Save new phase/order configuration(s) and reported metric(s) by calls
             to 'DseRun.updateBestSolution = function(newConfig, newFitness)'
        ...
  }
end
```

Figure 3.6: Example of a DSE algorithm template in LARA.

number of passes per sequence. DSE scheme specific parameters can be passed in an object holding pairs of parameter names and settings (for each parameter). The aspect implementing the DSE algorithm accesses an object that encapsulates the functions that compile and execute the function/program being optimized (to be called from inside the LARA DSE algorithm implementation). The conditions for accepting compiler pass selections/orderings generated during exploration are programmed in the logic of each specific DSE algorithm implementation using methods available by the DSE system.

Integration with the toolchain that is responsible for compiling the function/program to optimize with phase selection/ordering specialization and remaining files, the execution/simulation of the generated solution, and the extraction of the metric of interest (e.g., performance) is abstracted by a call to a function of the DSE system, which returns the value for the metric associated with the use of the phase selection/order passed as input to that function. This abstraction allows for runtime selection of the compiler toolchain to use (e.g., Clang/LLVM, GCC) without requiring maintaining multiple versions implementing the same DSE logic (e.g., one per compiler toolchain supported), thus facilitating the development of new DSE schemes.

3.4.1 Phase ordering with REFLECTC

The use of the REFLECTC CoSy-based compiler (Nobre et al., 2013a) in a LARA DSE loop allows the exploration of multiple compiler sequences for any of the supported targets (i.e., PowerPC, X86, MicroBlaze, VHDL). Additionally, support for other targets (e.g., other microprocessor ISAs, GPUs) can be added to the existing REFLECTC tool, as long as they are provided by ACE or other developers of CoSy backends.

The selection of the set of compiler passes for exploration in the DSE schemes is relevant. A "smart" selection of compiler passes will allow to reduce the exploration time while possibly not compromising the solutions achievable using the derived sequences. Table 3.2 depicts the compiler passes, called "engines" in ACE's CoSy documentation, used in our DSE experiments.

The REFLECTC compiler not only allows specifying which compiler passes to execute, and in which order, through LARA aspects, but it also allows a fine control of some of its compiler passes, such as loop unrolling.

3.4.2 Phase ordering with LLVM

Each iteration of DSE using LLVM executes the LLVM optimizer (*opt*) with the LLVM assembly representation of the function/program and a sequence of compiler passes to evaluate as input, resulting in a new LLVM IR representation of the program transformed by the compiler passes. This new code representation is then passed to the LLVM Static Compiler (*llc*), which generates assembly code to the chosen target architecture. Executable code for the target architecture is generated with a linker targeting that same target. We rely on the Clang frontend to generate, at the beginning of the DSE process, the unoptimized LLVM code representation of the input program/function, which is used by all the executions of the LLVM Optimizer called from inside the DSE algorithm logic programmed in LARA. Figure 3.7 shows the compilation flow when optimizing a given function using Clang/LLVM.

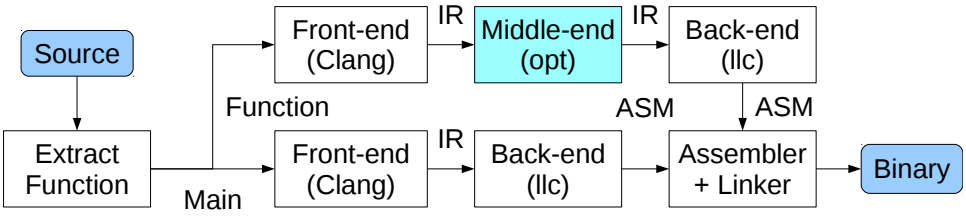

Figure 3.7: Compilation flow using LLVM (Reprinted from (Nobre et al., 2016b)).

We used Clang because it supports the C programming language. We could have used other frontend with our DSE system in case the sources of our test kernels were programmed, for instance, in Fortran. Unlike with REFLECTC, we do not use an integrated weaver. We rely on a command line interface to specify the compilation sequences.

Table 3.3 depicts the 73 LLVM compiler passes used by the LLVM OPT's –Ox flags (i.e., the 72 passes in –O3 plus the *always-inline* pass exclusive to –O1). LLVM –O3 uses all compiler passes in –O1 and –O2, and a number of additional passes.

Because of support to different microprocessor targets, we used different versions of LLVM (and a different sets of passes). The reason for this is that one of our targets, MicroBlaze, stopped being supported after LLVM version 3.3. MicroBlaze code generation in not part of the LLVM compiler anymore.

Table 3.2: REFLECTC compiler passes considered by DSE schemes. A list of 49 compiler passes were selected with input from ACE.

Pass Name	Description	Pass Name	Description
algebraic	Apply several algebraic simplifications.	loopive	Loop induction variable elimination.
alias	Set the Aliased field.	loopremove	Empty loop optimizer.
blockmerge	Merge basic blocks.	looprev	Rewrite loop to count down.
cache	Cache memory locations in local tempos.	loopscalar	Loop scalar replacement.
chainflow	Jump-to-jump optimizer.	loopstrength	Pointer expr. to variables.
ckfstrength	Calls cheaper known function alternatives.	lowerbitfield	Lower the operators mirBitExtract and mirBitInsert.
condassigncreate	Replace control dependencies by conditional assign structure	lowerboolval	Rewrite mirBoolVal to explicit control flow.
conevun	Constant evaluation.	lowerpfc	Translate mirPureFuncCallstomirFuncCalls.
constprop	Constant propagator.	lrrename	Life range renaming of local variables.
copyprop	Propagate copies of variables.	markconvert	Mark mirConvert when argument is in range.
cse	Common sub-expression elimination.	misc	Miscellaneous optimizations.
demote	Demote bit width of operations.	mvpostop	Move post-operator assignments.
dismemun	Lower to expressions consisting of integer arithmetic.	noreturn	Remove unreached code after not returning function call.
domorder	Order blocks with the help of the dominator tree.	promote	Arithmetic expression size promotion.
exprprop	Expression propagator.	rodata	Sets all non-aliased mirDataGlobals with static linkage to read-only.
funceval	Evaluation of f. calls using value range.	scalarreplace	Replace local structured variables by scalars.
globcse	Global common subexpression elimination.	setpurity	Compute side effect flags for procedure
hwloopcreate	Rewrite IR into HWLoop constructions.	setrefobj	Set the AbstractValue field of pointer objects.
ifconvert	Control dependencies by data dependencies.	strength	Strength reduction.
loopbcount	SetUseEstimateof blocks based on loop nest.	tailmerge	Merge common tails.
loopcanon	Canonization actions on loops.	tailrec	Eliminate tail recursion.
loopfuse	Fuse two adjacent loops.	vprop	Propagation of expressions using value range.
loopguard	Change while-do loops into protected do-while.	vshrink	Reduce size of local variables.
loophoist	Hoist statements from loops.	vstrength	Strength reduction based on value range.
loopinvariant	Invariant code motion.		

Table 3.3: LLVM Optimizer compiler passes used by the compiler sequences associated with the –Ox flags. Descriptions taken from LLVM OPT.

Pass Name	Description	Pass Name	Description
aa	Function Alias Analysis Results	licm	Loop Invariant Code Motion
adce	Aggressive Dead Code Elimination	loop-accesses	Loop Access Analysis
alignment-from-assumptions	Alignment from assumptions	loop-deletion	Delete dead loops
always-inline	Inliner for always_inline functions	loop-distribute	Loop Distribition
argpromotion	Promote 'by reference' arguments to scalars	loop-idiom	Recognize loop idioms
assumption-cache-tracker	Assumption Cache Tracker	loop-load-elim	Loop Load Elimination
barrier	A No-Op Barrier Pass	loop-rotate	Rotate Loops
basicaa	Basic Alias Analysis (stateless AA impl)	loop-simplify	Canonicalize natural loops
basiccg	CallGraph Construction	loop-unroll	Unroll loops
bdce	Bit-Tracking Dead Code Elimination	loop-unswitch	Unswitch loops
block-freq	Block Frequency Analysis	loop-vectorize	Loop Vectorization
branch-prob	Branch Probability Analysis	loops	Natural Loop Information
constmerge	Merge Duplicate Global Constants	lower-expect	Lower 'expect' Intrinsics
correlated-propagation	Value Propagation	mem2reg	Promote Memory to Register
deadargelim	Dead Argument Elimination	memcpyopt	MemCpy Optimization
demanded-bits	Demanded bits analysis	memdep	Memory Dependence Analysis
domtree	Dominator Tree Construction	mldst-motion	MergedLoadStoreMotion
dse	Dead Store Elimination	opt-remark-emitter	Optimization Remark Emitter
early-cse	Early CSE	pgo-icall-prom	Use PGO instrumentation profile to promote indirect calls to direct calls
elim-avail-extern	Eliminate Available Externally Globals	profile-summary-info	Profile summary info
float2int	Float to int	prune-eh	Remove unused exception handling info
forceattrs	Force set function attributes	reassociate	Reassociate expressions
functionattrs	Deduce function attributes	rpo-functionattrs	Deduce function attributes in RPO
globaldce	Dead Global Elimination	scalar-evolution	Scalar Evolution Analysis
globalopt	Global Variable Optimizer	sccp	Sparse Conditional Constant Propagation
globals-aa	Globals Alias Analysis	scoped-noalias	Scoped NoAlias Alias Analysis
gvn	Global Value Numbering	simplifycfg	Simplify the CFG
indvars	Induction Variable Simplification	slp-vectorizer	SLP Vectorizer
-inferattrs	Infer set function attributes	speculative-execution	Speculatively execute instructions
inline	Function Integration/Inlining	sroa	Scalar Replacement Of Aggregates
instcombine	Combine redundant instructions	strip-dead-prototypes	Strip Unused Function Prototypes
instsimplify	Remove redundant instructions	tailcallelim	Tail Call Elimination
ipsccp	Interprocedural Sparse Conditional Constant Propagation	targetlibinfo	Target Library Information
jump-threading	Jump Threading	tbaa	Type-Based Alias Analysis
lazy-block-freq	Lazy Block Frequency Analysis	tti	Target Transform Information
lazy-value-info	Lazy Value Information Analysis	verify	Module Verifier
lcssa	Loop-Closed SSA Form Pass		

In addition to the phase ordering experiments with the passes from the compiler sequence represented by the –Ox flags, we also did experiments with additional passes.

3.5 Validation of the solutions generated by DSE

In all the experiments presented here, functions that are passed as input to our phase selection/order exploration system are instrumented so that the outputs resulting from their execution can be compared with sets of expected outputs for each execution of a new optimized binary. Even when optimizing for other metrics such as code size, which would typically not require executing the resulting code/binary, any new executable code produced by the DSE system is executed for validation.

For benchmarks with floating-point data types, we classify as correct each output value if the absolute value of the difference between the two outputs (a positive floating point number) is not higher than 0.001.

3.6 Detection of problematic sequences

Some compiler sequence might be considered unfit for other reasons than generating non-functionally equivalent code (i.e., "wrong" code').

The compiler can halt because of bugs in some of its compiler passes. These bugs might be unnoticed for a long time. For example, a given compiler pass might always work as intended until some very particular program/function IR, result of a previous application of a certain sequence of other passes, is presented to it.

A problem with the compiler optimizer module (e.g., LLVM Optimizer tool) is reported when the optimized version of a given input function is not generated after a time limit, and a problem with the backend module is reported when no executable is generated after a time limit. Additionally, the use of some phase selections/orders might result in compiled code that is so slow that it takes longer than a time limit to execute. This values are set as part of the general exploration settings, but can also be set on a function-by-function basis. We argue that if this time limits are set to large enough values (i.e., at least slightly longer than the execution time for the non-optimized function/program), then any sequences leading to executables that exceed the time limit are not of interest in most scenarios. Specially when the most relevant metric is execution performance.

All of these situations are detected and reported by our compiler phase ordering exploration infrastructure. It is important to notice that the label of "problematic sequences" attributed to a set of sequences resulting in problems with the compiler optimizer or code generator is only valid in the context of the compiler toolchain used. Other version of the compiler toolchain might not have any problem with a number of the sequences classified as problematic in some instance of the compiler toolchain, even considering the same functions/programs.

3.7 General approach for reducing DSE execution overhead

While performing DSE experiments we achieved the conclusion that there are some best practices resulting in higher overall exploration efficiency that can be applied independently of the exploration algorithm, the compiler toolchain, and the target platform. They are realized by saving the measured fitness values (e.g., performance, energy, memory use), unique representations of the executable files (e.g., the binary/ELF files) generated from compilation with the sequences tested during phase selection/ordering exploration, and unique representations of the sequences themselves.

A DSE session includes a source code, a compiler toolchain, and a target platform. For a given exploration session a sequence is only evaluated a single time. This results in exploration overhead in cases the same sequences are generated multiple times in the same DSE session. This is accomplished by saving a lossless compressed representation of all the sequences evaluated during a DSE session, as well as the target metric measured for each evaluated sequence. If a given sequence generated in a given DSE iteration was already generated and evaluated previously, the DSE loop continues without incrementing the iteration counter. Most of the overhead of exploration is caused by the compilation and execution/simulation steps, so this effectively reduces the exploration time.

A DSE session from our phase selection/ordering system keeps information about what optimized function/program instances were already evaluated. If an optimized version of a given function produced by a new compiler phase order has the same digest as other optimized version previously tested in the same session they are considered to be equal, and for this reason a new evaluation (i.e., testing and validation) is avoided. Its fitness value and the result of validation for functional correctness would already be known at that point. Although the exploration process can avoid executing optimized versions of functions that are equal to other previously tested, it still increments the DSE iteration counter. Compilation time can represent a considerable percentage of the exploration overhead. Depending on the compiler, the compiler settings, the target, and the kernel and kernel parameters, it can even surpass the overhead resulting from the evaluation of the generated codes/binaries. We use SHA1, and therefore it is very unlikely that collisions happen by chance (i.e., different compiled code resulting in the same hash). In fact, because we are only interested in minimizing collisions that could happen by chance, even MD5, which is considered less secure cryptographically, can be used instead of SHA1 in systems where the implementation of the former is faster than the implementation of latter. Figure 3.8 represents how we avoid executing binaries that were already executed because a previously tested sequence producing the same output code. In the exploration instance depicted, *Code 1* and *Code 2*, generated by optimization with sequences *S1* and *S2*, respectively, result in the same digest message when hashed with SHA1, meaning that they are in fact identical codes. The same goes for *Code 3* and *Code 5*, the result of optimization with *S3* and *S5*, respectively.

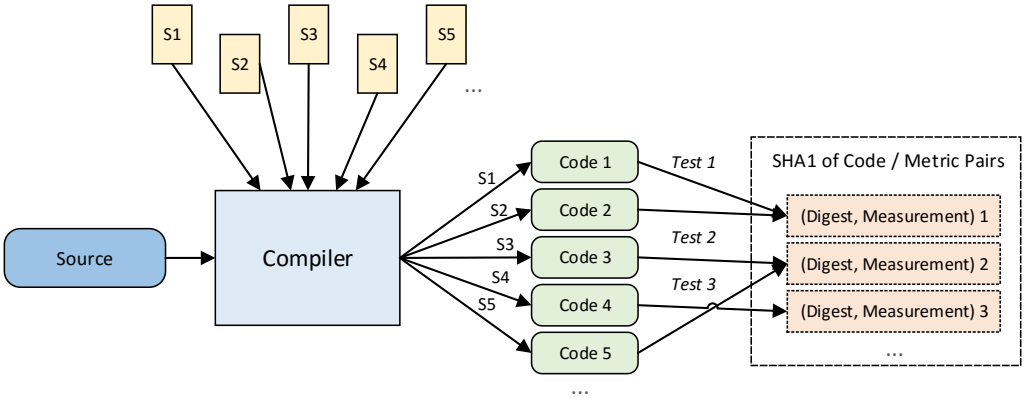

Figure 3.8: Reducing execution overhead by only executing different codes.

3.8 Using the DSE system

Our DSE system is called by a command line tool, *larad*. The command line tool is implemented as a script that calls the LARA interpreter with the main aspect of the DSE system, with the parameters passed to it by command line. The tool receives as input the following parameters: the name and location of the compiler to use (can be useful to experiment with different versions of a compiler toolchain), the target (e.g., ARM, LEON3, MicroBlaze, X86), the name of the DSE algorithm, the name of a file containing the list of compiler passes (or flags in case of performing phase selection only with, e.g., GCC) to consider for exploration, a maximum number of compilations/evaluations, the maximum sequence length for the generated sequences, a name identifying a compiler sequence cleaning strategy (performed after exploration with the DSE algorithm terminates execution), and the name of the file containing the code to optimize. Most of these parameters are optional. In case they are not set by the user, the DSE system assumes default exploration settings. The DSE command line interface can also optionally receive arguments to parameters that are specific to the DSE algorithm being used.

The DSE system searches for compiler sequences that better optimize the code passed as input to the command line interface. This file can contain only a function or multiple functions (e.g., the function to optimize might call other functions). All code to evaluate a given function can be in the file passed to the DSE system (i.e., includes the `main()` and all functions required), or it can be in a separated file or files. In the former case, the DSE system performs compiler phase selection/ordering specialization over the whole program. In the latter case it is able to perform function-specific specialization. The remaining code, which for the codes used in this thesis is in a single separated file, is compiled separately, resulting in object code that is later linked with object code generated for the optimized function(s). We perform function-specific specialization in all experiments presented in this thesis, thus only the function to optimize and functions that are called by it are defined in the file passed as input to the DSE system.

Figure 3.9 presents the command line that instructs our DSE system to target a LEON3 CPU (`-target=leon3`) using LLVM (`-compiler=llvm`), with a uniform random probing explo-

ration method (`-algo=random`), for 10,000 iterations (`-nsteps=10000`) and a sequence length of up to 16 compiler passes (`-seqlen=16`) selected from a set containing all and only the passes present in the compiler sequences represented by the compiler –Ox flags (`-passes=OX`), and using a specific approach for cleaning the final sequence (`-clean=sequential`). In this example no location to a specific LLVM toolchain is passed to the command line tool, so the default LLVM toolchain is used,.

```
larad -compiler=llvm371 -target=leon3 -metric=size -algo=random -nsteps=10000 -
    passes=OX -seqlen=16 -clean=sequential IMG_sad_8x8_c.c/IMG_sad_8x8_c.c
```

Figure 3.9: Example of utilization of the DSE infrastructure.

The condensed output resulting from execution of the command line depicted in Figure 3.9 is depicted by Figure 3.10.

For the function defined in `IMG_sad_8x8_c.c`, and using uniform randomized exploration, exploring compilation with 10,000 different compiler sequences, our DSE system finds a compiler sequence that results in the generation of code that is 1.83× smaller than the code obtained with the default compiler optimization levels, including –O0. For this particular kernel and target, the –Ox level that results in smaller code size is –O0, and therefore, when the exploration outputs the improvement over the –Ox levels, it is against –O0 that the code size of the optimized function resulting from the use of the sequence leading to smaller code size is compared to. While –Os is typically more concerned with code size than –O2/–O3, it still uses optimizations that can lead to increasing code size. The fact that the sequence reported under `Best Solution Found` has less than 16 compiler passes, is the result of cleaning the solution found by exploration to be most suitable for optimizing regarding the objective metric of interest.

Although this DSE instance considered 10,000 compiler sequences, which may be a large number for some kernels/targets, the DSE time was only 203 seconds because of our approach of evaluating only generated codes that are different from the codes generated up to a given exploration point. In this particular exploration instance, 9,642 out of the 10,000 generated optimized code implementations of the `IMG_sad_8x8_c` function did not need to be evaluated, thus considerably reducing the exploration overhead. This feature in part of our DSE system and, given it is not deactivated by the user, it works in any compiler pass exploration instance (i.e., with any supported target, metric, set of passes, compiler).

This exploration included only the flags used in the sequences represented by the –Ox compiler flags from LLVM. When including additional passes while maintaining all other exploration variables, our DSE infrastructure finds a sequence that results in code that is 2.2× smaller than any –Ox level (also including –O0). This is a considerable improvement over what was achieved using only the passes that are present in the –Ox sequences (2.2× vs. 1.83×).

```
---------------------- DSE CONFIGURATION -------------------------
target: leon3, compiler: llvm, algo: random, metric: code size (bytes)

Collecting Clang/LLVM Optimization Levels...
        -O0: 176, -O1: 272, -O2: 272, -O3: 272, -Os: 272

Exploring 10000 compiler pass sequences using Uniform Random Search algorithm while
     considering a maximum of 16 compiler passes per sequence...
        1 (O)    : 148   : -mldst-motion -globaldce -strip-dead-prototypes -loop-
            unroll -deadargelim -instcombine -deadargelim -bdce -indvars -
            instcombine -memcpyopt -instcombine -globalopt -instcombine -elim-avail
            -extern -gvn
        2 Problematic sequence (Optimized function IR not generated)     :
            1.7976931348623157e+308 : -float2int -argpromotion -functionattrs -
            constmerge -no-aa -assumption-cache-tracker -lcssa -loop-deletion -adce
             -globaldce -adce -lazy-value-info -loop-idiom -bdce -dse -domtree
        3 Problematic sequence (Optimized function IR not generated)     :
            1.7976931348623157e+308 : -loop-vectorize -assumption-cache-tracker -
            tti -no-aa -basicaa -loop-accesses -memdep -lcssa -licm -loop-idiom -
            domtree -always-inline -basiccg -sroa -verify -globalopt
        4 (X)    : 156   : -loop-rotate -jump-threading -adce -loop-simplify -tti -
            lazy-value-info -bdce -inline -slp-vectorizer -functionattrs -basicaa -
            elim-avail-extern -assumption-cache-tracker -loop-unswitch -
            tailcallelim -globalopt
        5 (O)    : 132   : -loop-simplify -gvn -indvars -assumption-cache-tracker -
            always-inline -mldst-motion -reassociate -functionattrs -branch-prob -
            bdce -mldst-motion -simplifycfg -dse -instcombine -assumption-cache-
            tracker -loop-vectorize
...
        10000 (X)        : 172   : -indvars -functionattrs -loop-unswitch -
            reassociate -loop-accesses -inline -block-freq -instcombine -loop-
            deletion -always-inline -instcombine -branch-prob -functionattrs -loop-
            deletion -reassociate -adce

        Best Sequence Found : 96    : -lazy-value-info -tailcallelim -lazy-value-
            info -loop-deletion -lower-expect -loop-accesses -tti -correlated-
            propagation -adce -gvn -sroa -tbaa -loop-accesses -loop-rotate -licm -
            basicaa

Best Solution Found (after cleaning)...
        Sequence : -sroa -lazy-value-info -loop-rotate
        Metric  : 96

Improvement Over Clang/LLVM Optimization Levels...
        Over -O0 : 1.83, -O1 : 2.83, -O2 : 2.83, -O3 : 2.83, -Os : 2.83
        Over Best -Ox : 1.83

Design Space Exploration Execution Stats...
        Number Of Iterations : 10000
        Exploration Time (in seconds) : 202.592
        Number of Binaries Saved From Testing :9642
```

Figure 3.10: Condensed DSE execution output.

3.9 Extending the DSE system

Our DSE system is organized in a way that support for new DSE algorithms, targets, compilers and metrics can be added using a configuration-based approach. It ships packaged with the distinct types of components under specific directories (e.g., `algorithms`, `compilers`, `platforms`). The DSE system can be extended by populating with more entries the directory that holds the type of components of the same kind one wants to develop. Alternatively, the user can extend the DSE system without having to modify the DSE system tree by using a feature of the LARA interpreter.

The LARA interpreter allows grouping components by declaring the contents of folders as LARA *bundles* of a given kind. The declaration is performed in a configuration file (`lara.bundle`) inside the root directory of each bundle. The folders `algorithms`, `compilers` and `platforms` in the DSE system tree are declared with a tag as bundles of the kind *algorithm*, *compiler* and *target*, respectively. The user can declare folders external to the DSE filesystem tree (i.e., folders not shipped with the DSE system) as LARA bundles and pass their locations to the LARA interpreter, making the contents of each included external bundle (e.g., a subdirectory with additional DSE algorithms) automatically accessible to the LARA runtime. Selection of which specific component (e.g., specific target) to use, from components tagged as being of a given kind, is performed with the `-bt` parameter of the LARA interpreter. For instance, passing `-bt compiler=llvm` by command line when executing the LARA interpreter instructs it to include the components from inside the folder(s) named *llvm* that are inside bundles/folders tagged as being of the type *compiler* (`tag=compiler`). In the DSE system tree, there is a folder named `compilers/llvm`, which includes LARA aspects and configuration files that are generic to the use of LLVM. The use of LARA bundles allows to import the intended LARA aspects at runtime as if they were in a single folder, while maintaining them separated for better code organization in addition to providing a mechanism that facilitates extension. The `-i` parameter can be passed to the LARA interpreter to make other LARA bundles visible to the DSE system (e.g., user has additional DSE algorithms in a folder outside the DSE system tree).

3.9.1 Compilers

Adding support for a different compiler toolchain is accomplished by: 1) adding a new folder, which name will identify the compiler in calls to the DSE system from the command line, under the `compilers` folder inside the DSE system tree, or in other folder passed to the LARA DSE system as an external bundle (`tag=compiler`); 2) creating configuration files identifying the passes and/or flags to explore and identifying the optimization strategies that will serve as baseline for the calculation of the improvements achieved with DSE (e.g., the standard optimization levels); and 3) creating the LARA code for implementing the compilation stages.

Adding a new compiler requires making the compiler tools available to the DSE system (e.g., for GCC this is in `compilers/gcc/CompilerSetup.lara`), and declaring the general way to use the tools for at least a source/language. For instance, for compiling C files with GCC, we have a general compilation interface that is specified in `compilers/gcc/c/Compiler.lara`. Other

general compilation interfaces can exist (e.g., `compilers/gcc/fortran/Compiler.lara`). This LARA file encapsulates the logic that implements the compilation of programs organized as previously stated, i.e., where the code of the function(s) one wants to optimize by specialized phase selection/ordering is separated in a different file (see Section 3.8). Functions declared in a given `Compiler.lara` are called when performing DSE with the compiler toolchain associated with that version of the aspect from inside the LARA aspect for the specific DSE algorithm selected at runtime.

Figure 3.11 depicts the generic code that makes the LLVM compiler tools available (`compilers/llvm/CompilerS` and Figure 3.12 depicts the code that, using the functions declared in the `CompilerSetup.lara` aspect, instructs our DSE phase selection/ordering exploration system on what to do in each of the different compilation stages. In the case of LLVM, function `Compiler.frontend()` is responsible for the generation of IR from the source file with the code to optimize, function `Compiler.opt()` calls the LLVM Optimizer tool with a compiler sequence, and function `Compiler.backend()` calls the LLVM static compiler to generate assembly code from the optimized IR generated by the previous step. Function `Compiler.compilemain()` compiles the remaining C code (outside the C code source file that includes the function(s) to optimize), and function `Compile.link()` links the assembly generated from the optimized function(s) with the remaining code (including the `main()`). In addition, an additional binary (`application_verification`) is also executed, in case verification is performed in a separated main. This option can be useful in cases where, for example, allowing verification in a separated `main()` makes it faster to refactor a given program so it can be used with the DSE system.

3.9.2 Platforms

Platforms are declared in folders inside LARA bundles (`tag = platform`), as it is with compilers, metrics, and algorithms. Each folder represents a target and includes subfolders that are also LARA bundles themselves. LARA files inside these subfolders are used to specialize the DSE steps to the target, including how metrics are measured and reported.

Inside the folder for each target, there is also a configuration file declaring the compilation interface/language used when compiling for this target, the default list of passes or flags to use for exploration and which parameters to pass to the functions that perform the compilation steps for the compilation interface (e.g., from `compilers/llvm/c/Compiler.lara` for LLVM). Each target is tied with a specific compilation interface/language. Figure 3.13 depicts the contents of such configuration file for a LEON3 target considering compilation with LLVM.

3.9.3 Metrics

A platform has DSE system modules/components for instrumenting the input programs/functions and reporting to the LARA DSE instance the metric of interest (e.g., binary execution performance). Platforms are capable of supporting different metrics, extracted by different JAVA classes, automatically imported by the DSE system (through the LARA interpreter include mechanism).

```
import lara.Io;
import lara.Platforms;
import LaradSetup;

var CompilerSetup = {};

CompilerSetup._getExecutableFile = function(executableName, executableFolder) {
  var exe = undefined;
  if(executableFolder === undefined) {
    exe = executableName;
  } else {
    exe = Io.getPath(executableFolder, executableName).getAbsolutePath();
  }
  if(Platforms.isWindows()) {
    exe = exe + ".exe";
  }
  if(executableFolder !== undefined && !Io.isFile(exe)) {
    throw "CompilerSetup._getExecutableFile: Could not find executable '"+exe+"'";
  }
  return exe;
}

CompilerSetup.getClang = function() {
  return CompilerSetup._getExecutableFile("clang", LaradSetup.getCompilerFolder());
}

CompilerSetup.getLlc = function() {
  return CompilerSetup._getExecutableFile("llc", LaradSetup.getCompilerFolder());
}

CompilerSetup.getOpt = function() {
  return CompilerSetup._getExecutableFile("opt", LaradSetup.getCompilerFolder());
}

CompilerSetup.getClangIncludes = function() {

  return "-I"+LaradSetup.getLaradFolder()+"/compilers/llvm/clang_includes";
}

CompilerSetup.getVerifyExeName = function() {
  return LaradSetup.getExeName("application_verify");
}
```

Figure 3.11: Code for making the LLVM toolchain available to the LARA environment.

```
import lara.Io; import lara.util.ProcessExecutor;
import LaradSetup; import CompilerSetup;

var Compiler = {};

Compiler.clean = function(params) {
  var files = Io.getPaths("./", LaradSetup.getExeName("application"), LaradSetup.
      getExeName("application_verify"), "function.optim.ll", "function.optim.s", "
      function.ll", "function.c", "main_wtiming.c", "main_wtiming.ll", "
      main_wtiming.s", "include.h", "info.dat", "*.dot");
  for(file of files) Io.deleteFile(file); }
Compiler.compilemain = function(params) {
  Io.deleteFiles("main_wtiming.ll", "main_wtiming.s");
  System.execute(CompilerSetup.getClang() + " main_wtiming.c " + params[0] + " -
      emit-llvm -S -o main_wtiming.ll");
  System.execute(CompilerSetup.getLlc() + " " + params[1] + " main_wtiming.ll -o
      main_wtiming.s");
  var verifyExeName = CompilerSetup.getVerifyExeName();
  if(LaradSetup.isVerificationSeparated === true) { // If verification is performed
       in a separated main source file
    Io.deleteFile(verifyExeName);
    new ProcessExecutor().setTimeout(LaradSetup.defaultTimelimit)
    .execute(CompilerSetup.getClang() + " " + params[2] + " main_verify.c function.
        c -fopenmp -lm -o " + verifyExeName);
  } }
Compiler.frontend = function(params) {
  Io.deleteFile("function.ll");
  System.execute(CompilerSetup.getClang() + " " + params[0] + " function.c -emit-
      llvm -S -o function.ll"); }
Compiler.opt = function(params) {
  Io.deleteFiles("function.optim.ll", "*.dot");
  new ProcessExecutor().setTimeout(LaradSetup.defaultTimelimit)
    .execute(CompilerSetup.getOpt() + " " + params[0] + " function.ll -S -o
        function.optim.ll"); }
Compiler.backend = function(params) {
  Io.deleteFiles("function.optim.s", LaradSetup.getExeName("application"));
  new ProcessExecutor().setTimeout(LaradSetup.defaultTimelimit)
    .execute(CompilerSetup.getLlc() + " " + params[0] + " function.optim.ll -o
        function.optim.s"); }
Compiler.link = function(params) {
    new ProcessExecutor().setTimeout(LaradSetup.defaultTimelimit)
    .execute(CompilerSetup.getClang() + " " + params[0] + " -o " + LaradSetup.
        getExeName("application") + " function.optim.s main_wtiming.s -lm"); }
```

Figure 3.12: Code that specifies what is performed at any of the DSE infrastructure compilation stages.

```
{
  "language":"c",
  "exporter_name": {
      "performance":["tsimexporter"],
      "size":["sizeexporter"],
  },

  "compiler" : {
    "gcc" : {
      ...
    },
    "llvm" : {
      "passes":"passes_v390_leon3.json",
      "compilemain_params":["--target=sparc-unknown-none-elf", "-march=sparc -mcpu=
          v8"],
      "frontend_params":["--target=sparc-unknown-none-elf"],
      "opt_params":[],
      "backend_params":["-march=sparc -mcpu=v8"],
      "link_params":["-mcpu=v8"]
        }
  }
}
```

Figure 3.13: Code that specializes the compilation stages for the LEON3 CPU and compilation interface using LLVM.

For instance, current Intel CPUs and the ODROID-XU+E (Hardkernel) support measuring energy consumption, while most target platforms supported by the DSE system do not (e.g., LEON3). Adding support to a new metric for a given target, is accomplished by adding an entrance to a configuration file inside a folder corresponding to the target and developing the LARA aspects specializing the code instrumentation steps needed to support a number of metrics (e.g., performance, energy consumption, code size). This configuration file specifies the name that identifies the metric (passed as parameter to the command line tool), the name of the metric exporter. The metric exporter is responsible to read reports generated by the execution of the compiled binaries (e.g., `elapsed time: 4343`, `energy consumed: 345`). In order for a metric exporter to perform anything when executed from inside a LARA environment, an entry with its name has to be added to the LARA interpreter tools configuration file (i.e., `tools.xml`) identifying what Java class and method to execute.

3.9.4 DSE algorithms

DSE algorithms are implemented by LARA files in folders existing inside bundles of the kind *algorithm*. The name of the folder that holds the LARA aspects implementing a DSE algorithm represents the name that identifies the DSE algorithm in the command line interface of the DSE system. Copying the folder of a DSE algorithm to a folder with a new name under a bundle of the kind *algorithms* results in a new entry being automatically selectable in the command line interface

of the DSE system. Then, the LARA files under the new folder can be modified to implement the new algorithm. Inside a LARA folder representing a DSE scheme there is a main LARA aspect that adheres to the interface of the DSE schemes in the DSE system, and possibly a number of auxiliary aspects imported by the former. Figure 3.14 depicts an implementation of a SA-based DSE algorithm.

3.10 Summary

In this chapter we described how the execution of specific compiler passes can be ordered using LARA and our REFLECTC CoSy-based compiler.

We show how REFELCTC can be internally controlled by LARA aspects through a LARA weaving engine, not only regarding the execution of ordered sequences of compiler passes, but also regarding where to apply an analysis/transformation compiler pass specific parameters, accessing information about the source code through a query mechanism.

We also explain how our DSE system interfaces with the LLVM toolchain to compile programs using a specific compiler sequence, and how the DSE is implemented relying on the LARA outer loop mechanism.

Additionally, we explained the general approach that we use to substantially reduce the exploration overhead of all DSE algorithms, and how the functions compiled using the specialized phase orders are verified for correctness.

Finally, we gave an example of the use of the DSE system, and explained how it can be extended to support new compiler toolchains, metrics, exploration schemes, and target platforms.

```
import DseRun;

aspectdef DseAlgo
  input exporeParams = [], maxIterations, maxWidth, additionalParams; end

  var startTempFactor = -1 / Math.log(0.5);     // 50% chance of selecting a
      sequence that is 100% worse than currently selected sequence
  var finalTempFactor = -0.00001 / Math.log(0.01);  // 1%  chance of selecting a
      sequence that is 0.001% worse than currently selected sequence
  var currentSolution = {};
  currentSolution.config = DseRun.getBestSolutionConfig().slice(0);
  currentSolution.fitness = DseRun.getBestSolutionFitness();
  var referenceFitness = referenceConfigsFitness[0];  var optimLevel = "";

  println("\nSetting Simulated Annealing Paramenters...");
  var temp = referenceFitness * startTempFactor;  var temp_min = referenceFitness *
       finalTempFactor;
  var alpha = Math.pow(Math.exp(1), Math.log(temp_min/temp)/iterations); //
      Calculates alpha based on parameters

  for(; temp > temp_min; temp = temp * alpha) {
  var newConfig = new Array();
    newConfig = currentSolution.config.slice(0);
    if(newConfig.length < maxwidth) {
      var rand_posConfigList = Math.floor(Math.random()*(currentSolution.config.
          length + 1));
      var newPass = exploreParams[Math.floor((Math.random() * exploreParams.length)
           + 0)];
      newConfig = currentSolution.config.slice(0,rand_posConfigList);
      newConfig.push(newPass);
      newConfig = newConfig.concat(currentSolution.config.slice(rand_posConfigList)
          ); }
    else {
      var rand_posConfigList = Math.floor(Math.random()*currentSolution.config.
          length);
      var newPass = exploreParams[Math.floor((Math.random() * exploreParams.length)
           + 0)];
      newConfig[rand_posConfigList] = newPass; }
    newFitness = dseRun.compile_execute_and_report(newConfig);
    if(DseRun.isSolutionValid(newFitness) == false) {
      DseRun.printIterationInfo(newConfig, newFitness, null);
      continue; }
    if(DseRun.isLeftFitnessBetterOrEqualThanRightFitness(newFitness,
        currentSolution.fitness)) {  // better solution
      currentSolution.fitness = newFitness;
      currentSolution.config = newConfig;
      if(dseRun.isLeftFitnessBetterOrEqualThanRightFitness(newFitness, DseRun.
          getBestSolutionFitness())) {
      DseRun.updateBestSolution(newConfig, newFitness); }
    DseRun.printIterationInfo(newConfig, newFitness, "O"); }
    else {  // worse solution
      var rand = Math.random();
      var delta_lat = Math.abs(newFitness - currentSolution.fitness);
      var val = Math.pow(Math.exp(1), - (delta_lat / temp));
      if(val > rand) {  // worse solution accepted
        currentSolution.fitness = newFitness;
              currentSolution.config = newConfig;
        DseRun.printIterationInfo(newConfig, newFitness, "XO"); }
      else {  // worse solution not accepted
        DseRun.printIterationInfo(newConfig, newFitness, "X");  } } }
end
```

Figure 3.14: SA-based LARA DSE algorithm implementation.

Chapter 4

Phase Ordering Exploration

Specialization of compiler phase orders can be efficiently accomplished by an automatic Design Space Exploration (DSE) approach. The DSE can deal with the difficulty to find phase selections and phase orders that result in better generated code. The difficulties are related to the large number of compiler passes to select from, and because compiler passes interact with each others, and their combined effect highly depends on code features in ways difficult to predict. Although compiler experts may know certain compiler passes are to be executed before other passes, the impact on other optimization passes in latter stages in the compilation is difficult, if not impossible, to predict with accuracy. This is the case even for compiler writers, and is exacerbated by the fact that even the smallest changes to a number of compiler passes (e.g., between different versions of a compiler toolchain) can lead to changes in the interdependencies between passes; making automatic DSE worth seeking.

The exploration of compiler sequences can be an intensive computing process. Iterative approaches tend to require a number of exploration iterations (i.e., compile and evaluate cycles) ranging from hundreds to millions (depending on program/function and simulator/platform) in order to find compiler sequences that result in suitable optimization regarding the given metric(s) of interest. Compilers typically provide tens to hundreds of compiler passes, and in many cases it may be beneficial to explore large compiler sequences including repetitions of the same compiler passes in different points of the compiler phase order.

Efficient automatic DSE of compiler pass phase selection and phase ordering can be of high importance, especially when stringent requirements are not satisfied by generic –Ox optimization options (e.g., –O2) and developers are otherwise required to modify the code and/or apply different optimizations. By applying specific compiler optimization phase orders there is the chance that fewer manual code modifications (if any) are needed. The existing approaches can be divided in approaches that may only be suitable in situations where the overhead of DSE can be amortized by a number of important factors (e.g., compile once and execute multiple times, mass market embedded systems, supercomputers), and in approaches that while in some cases do not reach as high optimization, are much faster to achieve and therefore more likely to be used by the typical compiler user.

The most suitable approach will depend on the type of compiler user. Compiler users can be separated in the following three distinct categories:

1. User that does not care about optimization at all. Some compiler users do not even know about the standard optimization levels.

2. Compiler user that does not care deeply about optimization. Expects compilation to be fast, and would likely only use DSE-based compilation if the quality of the resulting solution (e.g., performance, energy consumption, size of binary) is at least comparable with what is obtained with –O2/–O3 at worst case.

3. Developer of systems/applications with strict requirements. Cares deeply about optimization. Such user can be willing to tolerate a DSE overhead of hours/days if it means there is a chance that the generated binaries or hardware designs, in the case of software compilation and hardware compilation, respectively, are a few percentage points better in relation to the objective metric(s) of choice.

Given that the first type of user identified above does not even care about optimization, our concern is focused on the other two types of programmers/users. Expectations of users/programmers have to be taken into account. For instance, someone may not want to use an exploration system such as ours if it requires $100\times$ more resources to find a specialized compiler phase order (may require many compile/execute cycles) that result in a $1.05\times$ performance improvement over –O3. Other users may be very satisfied with such proposition.

Given that a considerable number of users that want to achieve more performance have the habit of at least evaluating of the –Ox compiler flags (e.g., –O1, –O2, –O3) when compiling their functions/programs, we consider that a system that only requires a few more evaluations (e.g., no more than 10) to find sequences that tends to result in more optimized code would be well received by a number of users/programmers that are not interested in more exhaustive approaches (e.g., hill climber with $1,000$ iterations).

In this chapter we present a number of approaches that we developed and evaluated in the context of this Ph.D., in an effort to more efficiently suggesting specialized compiler phase orders.

We present a Simulated Annealing (SA)-based approach with embedded automatic parameter specialization. In our SA-based approach, the T_{min} and T_{max} parameters are specialized at exploration time to better suit the function/program being compiled.

We also present a graph-based approach, where a graph represents legal transitions between passes and/or subsequences. This graph can be used to generate an arbitrary number of new compiler phase selections/orders, and alternatively, guide other algorithms, such as SA-based approaches. Connections between graph nodes are weighted to favor sub-sequences that are more likely to result in suitable compiler sequences. The graph resembles previous compilation sequences and can be built from compilation sequences previously achieved for a number of reference functions/programs. In addition, we suggest approaches to use function/program features with the graph-based approach, allowing to specialize the graph to the particular functions/programs being compiled.

Finally, we present an approach inspired by the work of Purini and Jain (2013), as an effort in the direction of designing DSE approaches that can appeal to users that require close to no DSE overhead while still achieving better optimization than what is achieved with the standard optimization levels.

4.1 Simulated annealing approach

Simulated Annealing (SA) (Kirkpatrick et al., 1983), which got its name from the physical process undergone by misplaced atoms in a metal when heated and then slowly cooled, is an effective and practical algorithm for optimization problems. It is especially suitable in cases where the design space is too large for an exhaustive approach, which is the case when exploring compiler sequences resulting from the combination of tens or hundreds of compiler passes.

SA is not memory intensive as it does not need to keep information about previously explored design points and can escape the trap of local optima by accepting worse solutions with a probability that can start fairly high and lowers as the algorithms gets closer to termination. This allows the exploration of more design space points that would otherwise not be explored. The probability a "bad move" is accepted decreases with each successive loop iteration of the algorithm. The algorithm accepts design points that are worse than the current one with a probability given by Equation 4.1, where $P(e, e_{new}, T)$ is a random number between 0 and 1.

$$P(e, e_{new}, T) = e^{-\frac{\Delta E}{T}} \tag{4.1}$$

The probability function is affected by the current temperature T and ΔE, the cost difference (i.e., energy distance) between the new and the currently accepted solution, e_{new} and e respectively, as in Equation 4.2.

$$\Delta E = e_{new} - e \tag{4.2}$$

In order to avoid ignoring global optima after reaching it, the annealing temperature $T(k)$ is lowered after executing each outer-loop of simulated annealing. Therefore, the probability a "bad move" is accepted decreases with each successive loop iteration of the algorithm; in the same fashion molecules of a metal have increasingly less freedom as the metal cools down forming crystals during the annealing process.

We developed a SA-based approach as a first effort to develop an approach to search for compiler phase orders efficiently. Pseudocode for our SA-based approach is depicted in Algorithm 2. Variables s, e and t represent the state, energy and temperature at a given iteration. Variable s_0 represents the initial solution configuration. Variables s_{best} and e_{best} represent the currently accepted configuration (i.e., a compiler sequence) and associated energy (i.e., measure of objective metric). The function $temperature(k)$ calculates the next temperature T, and $perturbation(s)$ determines the next solution (i.e., new compiler sequence) using one or more perturbation rules, rules that generate the next configuration by changing the one previously accepted. Function $E(s)$

evaluates the energy used by a given configuration s. When modeling an optimization problem to the simulated annealing process, the energy represents the parameter one wants the algorithm to improve (e.g., number of processor cycles needed to execute a function). In the context of our work, energy represents the latency (i.e., number of cycles) of a given solution when targeting a given computing platform.

Algorithm 2: *SA-based* METHOD

Input: Number of iterations (N), maximum number of passes per sequence
Output: Best optimization sequence found (s_{best})

$s \leftarrow s_0$
$e \leftarrow E(s)$
$s_{best} \leftarrow s$
$e_{best} \leftarrow e$
$k \leftarrow 0$
$T \leftarrow T_{Max}$
while $T > T_{min}$ **do**
  $T \leftarrow temperature(k)$
  $s_{new} \leftarrow perturbation(s)$
  $e_{new} \leftarrow E(s_{new})$
  if $P(e, e_{new}, T) > random()$ **then**
    $s \leftarrow s_{new}$
    $e \leftarrow e_{new}$
  end
  if $e_{new} < e_{best}$ **then**
    $s_{best} \leftarrow s_{new}$
    $e_{best} \leftarrow e_{new}$
  end
  $k \leftarrow k + 1$
end
return s_{best}

Figure 4.1 depicts how the SA-based approach relates to other approaches based on key differentiating aspects of phase selection and/or ordering approaches, thus contextualizing this approach in relation to other approaches from the state-of-art in the topic.

4.1.1 Temperature update functions

One can use one from a set of update functions as means to decrement the SA temperature, or even a combination of functions (e.g., consider linear decrements until a given iteration number, and then use logarithmic decrements).

Geometric temperature update (see Equation 4.3), linear decrement (see Equation 4.4) and logarithmic decrements (see Equation 4.5) can be used as update functions in a SA-based DSE scheme. For the experiments presented in the next sections we use the geometric update function.

$$T_{n+1} = \alpha T_n \tag{4.3}$$

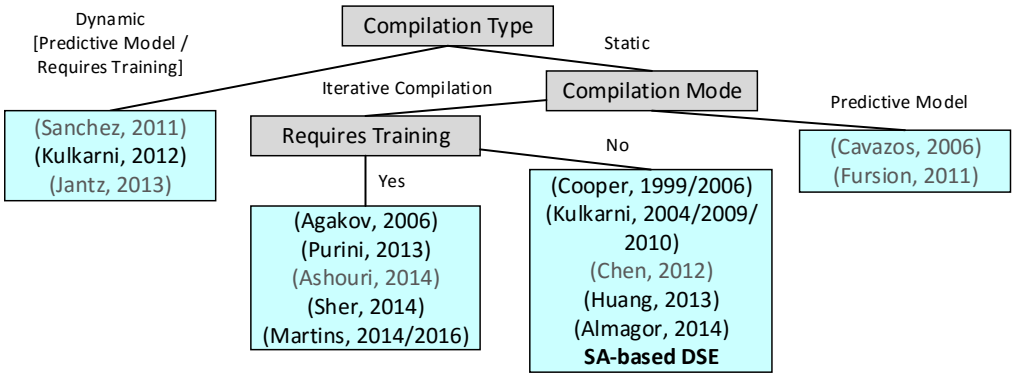

Figure 4.1: Contextualization of the SA-based approach in relation with approaches from the Related Work.

$$T_{n+1} = T_n - \beta \tag{4.4}$$

$$T_{n+1} = \frac{T_0}{\log T_n} \tag{4.5}$$

4.1.2 Selection of the SA parameters

We experimented with two different strategies for the selection of suitable values for these parameters. In a first version of the SA-based DSE algorithm, and using α (a multiplication factor) to calculate T_{n+1} given T_n, the selection of the values for T_{min} and T_{Max} was based on experimentation. With those parameters fixed, the value for α determines the number of steps. Thus, on this first version of the SA-based DSE scheme we allow the user to set α or the number of steps (one determines the other).

We later adopted a strategy where the T_{min} , T_{Max} parameters are automatically set at start (i.e., at runtime) of the execution of the DSE scheme. For instance, if execution performance is the exploration metric, then the input program is executed/simulated at start after compilation without optimization. The reported fitness on the target architecture is used to determine T_{min} , T_{Max}, using Equation 4.6, where P (value from 0 to 1) represents the chance given by the SA algorithm to accept a compiler sequence that is at most $W * 100$ percent worse than the currently accepted compiler sequence.

$$T_{Max/min} = -W * log_e P * fitness \tag{4.6}$$

Our current LARA SA-based DSE scheme, which is the one used for the experiments presented in the next chapter, is configured with the P_{Max} , P_{min} , W_{Max} and W_{min} and S parameters. Suitable values for these parameters were experimentally determined, analyzing the efficiency of

our SA-based scheme (i.e., number of iterations required to achieve a solution of a given quality) with different ranges for these parameters.

For SA-based exploration using a geometric update function, the α value, a SA parameter used in each iteration to calculate the current temperature by multiplication with the previous temperature, is calculated using Equation 4.7 as follows, after T_{min} and T_{Max} have been set at runtime.

$$\alpha = e^{\frac{\log_e \frac{T_{min}}{T_{Max}}}{S}} \tag{4.7}$$

4.1.3 SA transformation/perturbation rules

SA-based DSE relies on a set of perturbation rules to iteratively change the compiler sequence. Perturbation rules are applied by the perturbation(s) function in Algorithm 2. In each new iteration of the SA-based algorithm, a perturbation rule produces a new compiler sequence by performing changes on the current candidate compiler sequence. We explain next two perturbation rules implemented in our SA-based DSE scheme.

Perturbation *rule 1* consists in replacing a compiler pass from the currently accepted compiler sequence by a compiler pass from the considered list of compiler passes. Figure 4.2 represents an instance of SA using the REFLECTC compiler where the application of perturbation *rule 1* resulted selecting position number 2 from a candidate sequence with 6 compiler passes (numbered from 1 to 6) and replacing the compiler pass that was there (*-loopinvariant*) with a new compiler pass (*-loopguard*) from the list of compiler passes considered for exploration (e.g., passes in Table 3.2).

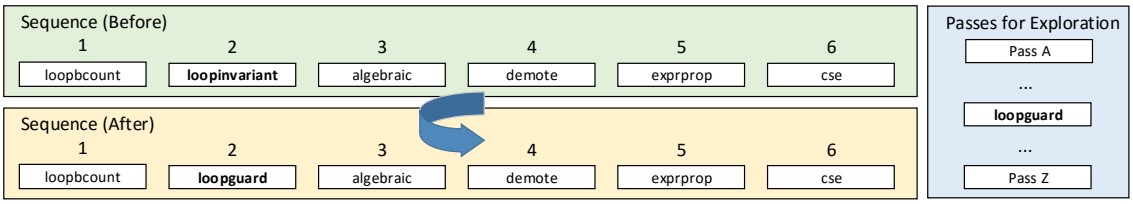

Figure 4.2: Example of application of SA perturbation rule number 1.

Perturbation *rule 2* consists in swapping two compiler passes randomly selected with equal probability from the currently accepted compiler sequence. Figure 4.3 represents an instance of SA using the REFLECTC compiler where the application of perturbation *rule 2* results in swapping compiler passes in positions 2 and 5, i.e., the *-loopinvariant* and *-exprprop* compiler passes.

In the case of the experiments presented in the following sections using the SA-based DSE scheme, the next optimization sequence to be evaluated for execution performance is generated in each iteration by a variation of perturbation/transformation *rule 1*: insert a new compiler pass (from the set of compiler passes to consider for exploration) in a random position of the current candidate compiler sequence, in case the candidate compiler sequence is smaller than a maximum

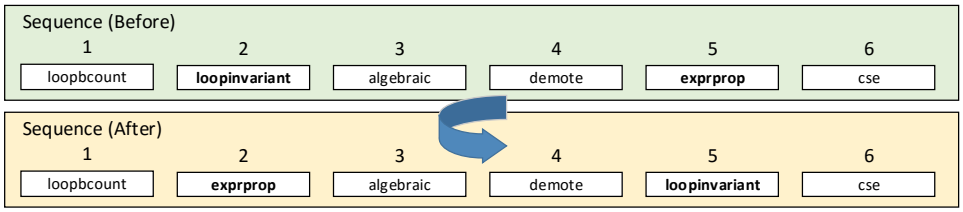

Figure 4.3: Example of application of SA perturbation rule number 2.

considered length; or replace a compiler sequence in a random position from the candidate compiler sequence with a new compiler pass, otherwise. Again, all positions in the compiler sequence, and all compiler passes from the considered compiler passes list, have equal probability of being selected.

4.2 Graph-based phase selection and ordering exploration

Knowledge about what compiler passes to use and their order of execution, so non-functional requirements can be best met, can be leveraged to develop more efficient DSE methods in the context of exploration of compiler sequences.

Our approach relies on a directed cyclic graph $G(V,E)$, where each vertex V represents a compiler pass (or a subsequence) and each edge E represents a transition between two compiler passes (or a subsequences). The graph can be used to represent favorable compiler pass transitions to more efficiently generate new compiler sequences. With this approach only a fraction of the compiler sequences space is considered for iterative compilation/execution, thus it has the potential to make exploration of compiler sequences faster. Paths in the graph formed by vertices connected through directed edges represent subsequences of compiler passes. These edges have weights, so that some subsequences of compiler passes are favored over others when generating new sequences. The use of the graph to guide an iterative compiler sequence exploration scheme has the potential to cause a reduction of the search space, by avoiding to consider a large number of compiler sub-sequences and giving preference to the ones that can be represented in the graph by paths associated with large weights. Figure 4.4 depicts how such approach relates to other approaches from the state-of-art.

4.2.1 Building the graph

The graph can be built by compiler experts based on their knowledge about compiler pass interdependence. Alternatively, in case far reaching human expertise about the nuances of compiler pass interdependence is not available, a statistical approach can be used. The graph can be built from a list of compiler sequences, each composed of two or more compiler pass instances (can include repetitions) ordered in a particular way, previously found for a group of functions/programs when compiling for the same or a similar target platform (e.g., CPU with similar architecture).

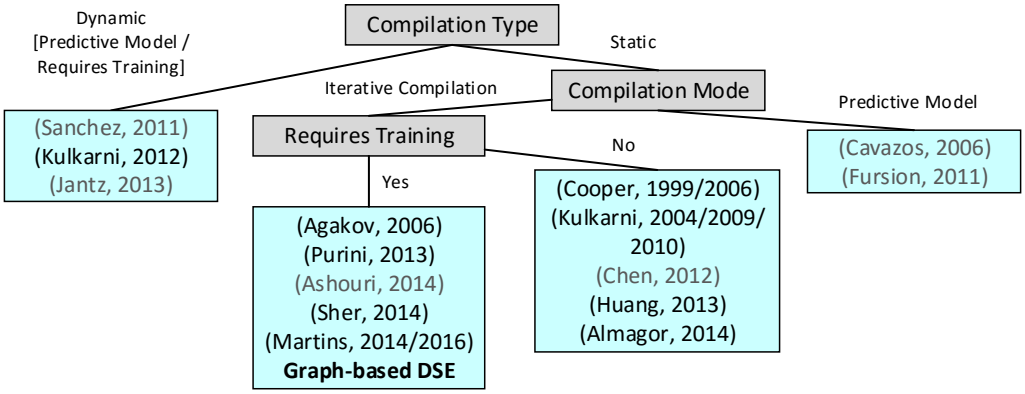

Figure 4.4: Contextualization of the graph-based approach in relation with approaches from the Related Work.

Building the graph from sequences previously found (e.g., with iterative compilation) for other functions/programs has potential as an approach. Given an objective metric, if some orderings of compiler passes are useful when compiling a set of programs/functions, then their use will more likely result in high quality solutions when used for optimizing new program(s)/functions(s) than other phase orders randomly selected from the complete search space. The list of compiler sequences is translated into the graph by providing graph paths according to those sequences and by assigning weights to connections between nodes. These weights can be based on how many times the pairs of compiler passes (or subsequences) represented by the connected nodes are present in the sequences used as input for the graph construction.

4.2.2 Example of building a graph from compiler sequences

The graph can be built using a set of sequences previously found using other DSE schemes, or even from subsequences manually devised using compiler expertise. For example, consider the following six LLVM compiler sequences (execution of passes is performed from left to right):

- *-instcombine*, *-mem2reg*, *-loop-rotate*
- *-gvn*, *-loop-rotate*, *-mem2reg*
- *-loop-rotate*, *-mem2reg*
- *-loop-rotate*, *-sroa*, *-indvars*
- *-loop-rotate*, *-instcombine*, *-licm*
- *-loop-rotate*, *-mem2reg*, *-indvars*, *-gvn*

These sequences were previously found to be suitable to compile functions from Texas Instruments (see Section 5.1.1) to a LEON3 processor. Typically, if using sequences previously found, there should be a concern in selecting sequences for functions/programs that are representative

of the domain of other functions that we want to optimize later leveraging the information stored in the graph to generate new compiler sequences. Figure 4.5 depicts the snapshots of the graph while being built from these six compiler sequences. In this graph instance, each node represents a single compiler pass. Each step represents the configuration of the graph after adding the information about compiler pass pairs in the sequence considered in the corresponding graph building step (out of the six steps). This method of constructing the graph is invariant with the order in which compiler sequences are considered (i.e., which is used first, second, and so on, does not change the resulting graph).

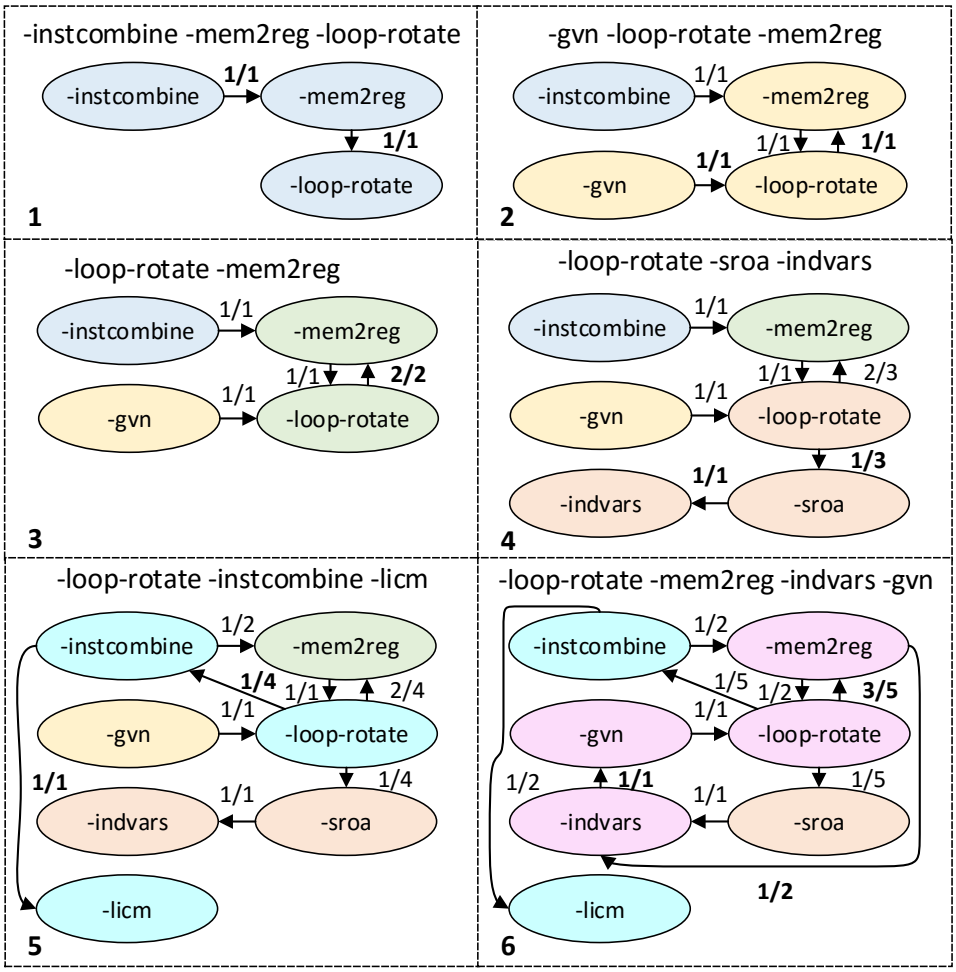

Figure 4.5: Representation of the construction of a graph from compiler sequences.

While this example considers pairs of compiler passes as the unit of information to be added to the graph, and our experiments presented in this thesis use this pair-wise modeling, the approach can be used with triplets of quads of passes as the base information unit. The vertices of the graph can also represent subsequences of variable length, which could result, for instance, from collapsing vertices in a previously built graph.

4.2.3 Dealing with passes with parameters

For the experiments with graph-based approaches presented in this thesis, we use the default LLVM implementation for all the compiler passes. For instance, for loop unrolling we do not specify the unrolling factor and let the LLVM compiler select the unrolling factor to apply to a specific loop using its internal heuristics. Therefore, we do not need to account for compiler passes with behavior modulated by different configurations in our experiments.

Intuitively, an approach to allow the exploration of compiler sequences using the graph while exploring at the same time different parameters in some of the compiler passes considered for exploration would be to represent the same pass in the graph as many times as the number of parametric configurations considered. However, we identified two problems with this approach. First, it could result in the generation of very large graphs, which would potentially loose expressive power. Second, it could result in the graph being too specialized to the particular programs/-functions used during the building phase of the graph, either using an automatic approach (e.g., compiler sequences previously found with other DSE schemes) or manual approach with knowledge about the compiler. We are still evaluating approaches for better dealing with this challenge, which will be specially more useful when the integration of our DSE system with LLVM allows for a more fine-grained control over the individual compiler passes.

4.2.4 Dealing with loops

The graph can contain loops, allowing compiler pass transitions (including subsequences of a number of passes) to appear subsequently in the compiler sequences generated from the graph. For example, considering the graph from Figure 4.5 in its final configuration, there are a number of allowed compiler sequences that include repetitions of the same pass/passes. The subsequence *-instcombine*, *-mem2reg* and *-looprotate* can repeat as many times as allowed by the exploration algorithm parameters concerning this behavior. A smaller repeating subsequence can be *-mem2reg*, *-loop-rotate*, or its reverse, *-loop-rotate*, *-mem2reg*, and a subsequence larger than both is *-instcombine*, *-mem2reg*, *-loop-rotate*, *-sroa*, *-indvars*, *-gvn*, *-loop-rotate*. Even larger repeating subsequences can be produced, e.g., from combining the previous subsequences.

A number of parameters can be used in a graph-based DSE scheme to deal with loops. For instance, the maximum number of times a given subsequence is allowed in a sequence and the maximum length of repeated subsequences. If the former parameter is set to 1 and/or the latter is set to 0 then no repetitions of subsequences are allowed.

Given how the graph stores information about valid transitions, conformance to these parameters is best checked during generation of new sequences from the graph. Building these restrictions into the graph itself by not allowing loops, could result in not allowing for the existence of a number of potentially meaningful connections. It could limit considerably the solution space in a non-meaningful way. For instance, considering the graph from Figure 4.5, if loops were not allowed in the graph itself as a measure to not allow the repetition of subsequences in new sequences generated from the graph, then the connection in the direction from *-loop-rotate* to *-mem2reg* and

its inverse connection could not both exist in the same graph. Similarly, in the case of the other loops with length greater than 2 (i.e., more than 2 graph vertices in the path forming a loop), at least a connection in the path resulting in the loop would have to be removed in order to break the loop.

4.2.5 The IterGraph approach

Algorithm 3 presents pseudo-code for an approach we call IterGraph. This approach is based on sampling a graph representing possible compiler pass sequences by vertices and weighted edges.

Algorithm 3: *IterGraph* METHOD

Input: Number of iterations (N), maximum number of passes per sequence (M) and graph (G)
Output: Best optimization sequence found ($bestSeq$)

```
1  bestSeq ← {}
2  bestFit ← getStartFitness()
3  for i ← 1 to N do
4      newSeq ← selectFirstNode(G)
5      for j ← 1 to M do
6          newPassOrSubseq ← nextNode(G, last(newSeq))
7          if isNull(newPassOrSubseq) then
8              break
9          end
10         newSeq ← append(newSeq, newPassOrSubseq)
11     end
12     newFit ← evaluate(newSeq)
13     if isNewSeqBetter(newFit, bestFit) then
14         bestSeq ← newSeq
15         bestFit ← newFit
16     end
17 end
18 return bestSeq
```

The algorithm receives as input the number of iterations N, the maximum number of passes M, and a graph G. A new sequence is generated and evaluated (i.e., compile and execute/simulate) in each iteration, starting with a call to $selectFirstNode(G)$ (line 4) to get the first compiler pass/subsequence, represented by a graph vertex, to add to a new sequence while being generated. For the experiments presented in this thesis, and unless stated otherwise, we start a new sequence by selecting the graph's most connected vertex. Other heuristics can be used; e.g., select the pass/subsequence most frequently used in the start of previously generated sequences or randomly select with probability of selection of each pass/subsequence based on its frequency in the start of previously generated compiler sequences.

Then, new compiler passes/subsequences are added to the current candidate sequence by following the graph edges using function $append(newSeq, newPass)$ (line 10) until the sequence is

composed of the maximum allowed number M of compiler passes; or until $nextNode(G, last(newSeq))$ (line 6) does not return a new pass/subsequence; which happens if there are no edges in the graph from the vertex last visited to other vertices. In case the graph was build from a set of compiler sequences (see Section 4.2.2), this represents the situations when a compiler pass/subsequence is always in the rightmost/last position of those compiler sequences.

The function $nextNode(G, last(newSeq))$ selects a new vertex by generating a random real number from 0 to 1; looking at the weights of the edges from the vertex last visited to other vertices and selecting the next vertex based on the random real number and a probability distribution determined by the weights.

Function $evaluate(newSeq)$ (line 12) evaluates the new sequence by compiling and testing (i.e., executing/simulating) a given program/function and returns a fitness value (e.g., CPU cycles for execution, energy consumption). Given an objective function (independent of the DSE algorithm and defined/selected by the user), $isNewSeqBetter(newFit, bestFit)$ (line 13) returns $true$ if the fitness value for the new sequence is better than $bestFit$ (the best fitness value until this point), and $false$ otherwise. If $isNewSeqBetter(newFit, bestFit)$ is evaluated to $true$ then the new sequence $newSeq$ and the fitness value $newFit$ are stored as the best sequence, $bestSeq$, and the best fitness value, $newFit$; respectively. At termination, the algorithm returns $bestSeq$, the best found sequence (line 14). The function $getStartFitness()$ (line 2) returns a very large positive or negative number, depending on if the purpose of the DSE is to minimize or maximize the fitness value $bestFit$, respectively.

4.2.6 Example of compiler sequence generation with IterGraph

Let's suppose during sequence generation (*for* loop in line 5 of Algorithm 3) from a graph representing only individual compiler passes in its vertices the sequence being generated is currently composed of the following compiler passes in the following order: *-loop-rotate*, *-basicaa*, *-loop-reduce*, *-loop-unroll*, and that each node represents a single compiler pass. Then, when selecting the next compiler pass to append to the sequence, the graph is inspected in order to suggest the compiler pass to be executed after the *-loop-unroll* compiler pass. Figure 4.6 shows part of an example graph representing the *-loop-unroll* node and the nodes it connects to. The weights of the connections determine the probability a given compiler pass is chosen to be appended to the compiler sequence after the *-loop-unroll* compiler pass. The next compiler pass to append to the compiler sequence being generated depends on a random number between 0 and 1 and the probability distribution generated from the weights of the out edges connecting the *loop-unroll* node to other nodes, depicted in Figure 4.7. As an example, if the randomly generated real number equals 0.75, then the *-indvars* compiler pass is selected; resulting in the sequence: *-loop-rotate*, *-basicaa*, *-loop-reduce*, *-loop-unroll*, *-indvars*.

Figure 4.8 depicts the steps for generating a complete compiler sequence of 4 compiler passes using the graph represented by Figure 4.5. Supposing that the heuristic for selecting the first pass chooses *-loop-rotate* as first pass, a new sequence is constructed form that graph as follows. The DSE process selects *-loop-rotate* as first pass for the new sequence, randomly selects a real

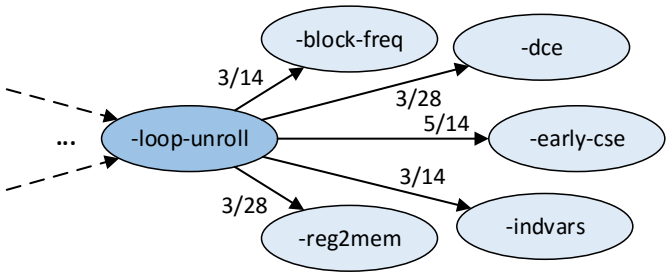

Figure 4.6: Representation of graph nodes and connections from node representing the *loop-unroll* compiler pass.

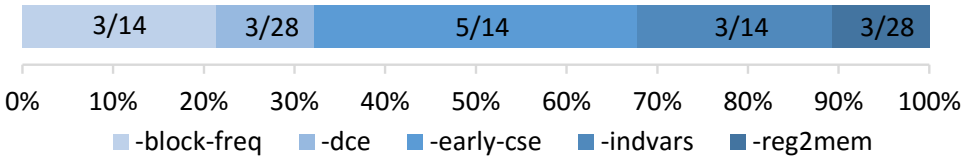

Figure 4.7: Probability distribution for next compiler pass selection considering the loop-unroll node as the current pass and the graph example shown in Figure 4.6.

number between 0 and 1, and checks where it falls in the probability distribution determined by the weights of the connections from *-loop-rotate* to nodes representing other passes. The weights of the connections determine that if the number belongs to the interval $[0, 1/5[$, $[1/5, 4/5[$ or $[4/5, 1]$ then *-instcombine*, *-mem2reg* or *-sroa* is respectively selected as next pass. In this example the number randomly generated takes a value between 0 and $1/5$, thus *-instcombine* is select as next pass. Supposing that the repetition of the same steps results in adding *-mem2reg* and *-loop-rotate* as third and fourth passes, respectively, then the final sequence would represent the execution of *-loop-rotate*, *-instcombine*, *-mem2reg* and *-loop-rotate*.

4.2.7 The SA+Graph approach

We developed an additional DSE approaches that rely on the graph. SA+Graph is an example of such approach.

In the SA approach previously introduced, when replacing or inserting a pass, i.e, during the initial phase of exploration when the maximum allowed compiler sequence length is yet to be reached, the algorithm randomly selects a compiler pass from the list of compiler passes considered for exploration. With information about what compiler passes/subsequences usually work well immediately before or/and immediately after a given pass/subsequence (represented in the graph), the replacement/insertion on any given position of the current candidate compiler sequence (the one that is currently accepted by the iterative exploration of SA) can be disregarded if the transition from the pass/subsequence immediately after that position and/or the transition to the pass/subsequence immediately after that position is/are not represented in the graph. One of the

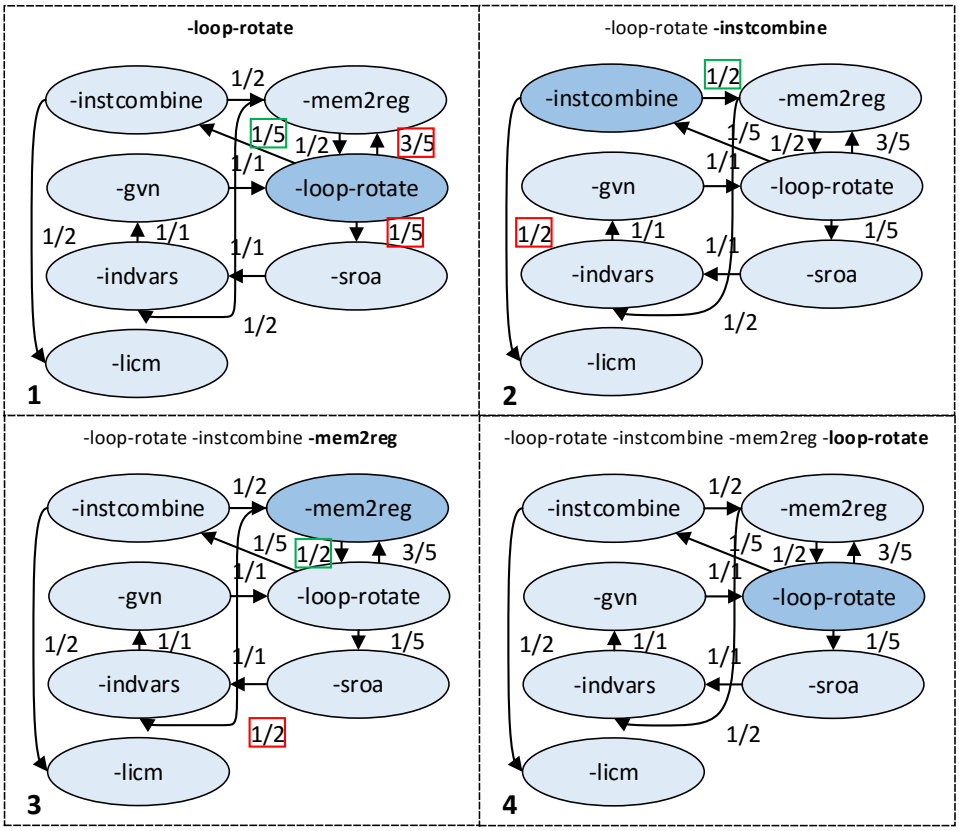

Figure 4.8: Generation of a compiler sequence using the IterGraph approach.

allowed compiler passes/subsequences at the insertion/replacement position is randomly chosen, while respecting the probability distribution determined by the graph out-edges.

4.2.8 Advantages over other DSE approaches

The fact the graph represents singular compiler passes and/or subsequences of passes has the potential to make it fundamentally more appealing to a number of users. Users/Programmers can visualize sections of the graph and grasp what are the configurations of sequences it can generate. This particularity of this method is not shared by many other methods (e.g., a number of machine learning methods, such as SVMs, random forests, and neural networks).

With this approach, at any given moment, programmers/users with knowledge about the compiler optimizer and backend (e.g., compiler writers) can observe a given graph snapshot and make modifications based on the new subsequences they want future graph-based DSE runs to allow and/or prioritize.

4.2.9 General challenges and limitations of enumeration-based approaches

The exploration of compiler sequences based on iterative approaches typically requires multiple compilations and simulations/executions of the program or parts of the program one wants to improve regarding the given objective metric(s). The simulation and/or execution of some programs/kernels can take "too long" with representative input parameters, thus posing a challenge to searching for compiler phase orders for some applications. Given a program/function, the exploration time when using a given iterative optimization phase order exploration algorithm is proportional to the time it takes for a single compilation and simulation/execution step. Approaches to reduce the simulation/execution time can be used (out of the scope of this work), such as the use of a smaller but still representative set of input program/function parameters or the use of techniques that estimate metrics (e.g., performance, energy, power) based on statistical analysis of the source code or an intermediate representation (IR) generated from it.

For some programs/kernels, the configuration of the suitable optimization phase orders may change with the input parameters. Such examples include programs/kernels with execution flow depending on input parameters, or programs/kernels that have a different performance depending on the shape of the input parameters. The use of specialized optimization phase orders found when using a set of input parameters may not result in the same improvement over baseline optimization levels when executing the program/kernel with a different set of input parameters. In the case different inputs change the way the application behaves (i.e., the number of computations of each kind) and/or how effectively the platform where it executes is used (e.g. because of caches), then it might be recommended to search for different compiler phase orders for each relevant case.

Compilers must generate functionally equivalent code. When specialized compiler pass sequences are used, there is always the risk of bugs caused by any given compiler pass, which may not have been previously detected by the battery of tests performed by the compiler developers.

We experimentally found that this is specially the case when generating large optimization sequences. Without going into formal verification of the optimized function/program by asserting its equivalence to a non-optimized version of the function/program, at the IR or assembly level, the only option left is to verify the output of the optimized program/function for a set representative of input parameters that would be used in a real use-case. Although we currently only use the latter approach, we are looking into possible formal verification approaches.

4.2.10 Limitations of the graph-based approaches

The IterGraph approach is more likely to generate sequences that result in wrong compiled code, a compiler crash, or in exceeding the imposed execution/simulation time limit, than the other DSE approaches tested. We experimentally found that when compiling some functions, the number of generated sequences that lead to one of those situations (which we are able to detect using our DSE system) is considerably higher than the sequences generated by any of the other algorithms. As an example, when exploring sequences for the *DSP_blk_move* function with 100,000 iterations, the IterGraph approach generated 29,168 sequences with problems (close to 30%), while that number was 17, 1,173, 1,499 and 5,046, for sequential insertion, GA, SA and SA+Graph, respectively.

Considering a given number of algorithm iterations, we look at this as an opportunity to improve the quality of the sequences (i.e., results in code better optimized regarding the given objective function) generated by our graph-based approaches by detecting sequences resulting in errors and preventing wasting resources by not evaluating them. As compilation and simulation take most of the resources in the DSE of compiler sequences that rely on those two steps (the execution of the DSE algorithm logic is virtually computationally insignificant when comparing with the compilation or simulation steps), the avoidance of compile/simulation steps for sequences that do not pass validation is important and will allow the DSE to explore additional sequences given the same timing constraints.

4.2.11 Suggestions to focus exploration with features

Considering a given target architecture and an objective metric, it is the source code of a given program/function and its inputs that determines what compiler sequences are the most suitable. Extracting features from the program/function and using them somehow to reflect on the compiler sequences that are generated by a given DSE algorithm has the potential to make exploration more efficient; i.e., result in suitable solutions being found faster, even if sometimes at the cost of loosing in terms of solution quality to a similar approach minus the code features when considering the latter with a large number of exploration iterations.

We do not present results in this thesis of experiments that use static and/or dynamic features in functions/programs to focus exploration with the graph-based approaches. Nevertheless, we suggest ways to achieve the integration of features in the graph-based exploration approaches, some of which are currently being evaluated.

Devising an approach that relies on features to achieve better DSE execution performance (than without features) can be a considerable challenge. If function/program features are poorly selected and/or poorly used to influence what compiler passes and/or sequences can be explored (or which are given priority) then a negative impact on exploration efficiency is a very likely possibility. Compiler sequence exploration including compiler phase ordering in addition to phase selection presents an additional challenge because of the much larger solution space. As a consequence, most approaches that consider static and/or dynamic features do it only the context of phase selection. For instance, to select a reduced set of compiler passes to consider for phase ordering exploration (Martins et al., 2016).

One of the most straightforward ways to use features in a context where the graph is built from compiler sequences found for a given reference set of programs/functions (see Section 4.2.2) is the following. Given a new program/function, identify based on features what programs/functions in the reference set of programs/functions are most similar with it. Then, instead of using sequences associated with all functions/programs used to generate a particular graph configuration, use only the sequences that were found to improve the codes that are most similar with the new given function/program. Depending on what functions are most similar, weights of the connections between vertices are strengthened/weakened and the number of vertices (i.e., compiler passes/subsequences represented in the graph nodes) can decrease compared with the graph built from all sequences. Only compiler passes and/or subsequences and transitions present in the sequences associated with the functions/programs that are most similar would appear in the new graph. This has the effect of focusing exploration, while hopefully not loosing too many suitable solutions from the reduced exploration space.

Identifying the most similar codes requires a similarity metric. Our experiments in the context of compilation of OpenCL code targeting a GPU using the cosine distance of feature vectors of MILEPOST (Fursin et al., 2008) features as similarity metric have shown promising results (see Appendix C). In the context of that work, given a new code, we rely on the cosine distance to identify the most similar OpenCL code from a set of reference codes, and then we use a sequence associated with that code (previously found by iterative exploration) to compile the former. The algorithm proceeds to find the next most similar codes and compiling/executing for as many iterations as the programmer/user can afford. The use of the cosine distance as similarity metric resulted in optimizing the OpenCL codes with considerably less compilations/executions than randomly selecting OpenCL codes from the reference set and using the compiler sequences found to be suitable for them. The results seem to suggest that there is some merit to the MILEPOST features used and to the use of cosine similarity as distance metric. In addition, the codes that are classified as being most similar using the cosine distance tend to look relatively more similar than others by manual inspection.

For the selection of the most similar functions/programs we envision a number of possible approaches. An approach (introduced above) consists in selecting sequences associated with the most similar functions/programs from a reference set. Alternatively, instead of selecting the sequences associated with a predefined number of functions/programs, considering sequences asso-

ciated with all functions/programs that are at least similar to a given degree (e.g., cosine distance equal or larger than 0.9) with the new function/program can have advantages and disadvantages over the former approach. For instance, always selecting a predetermined number of functions/programs that are most similar with the new function/program can have the downside of resulting in the use of sequences in the generation of the graph that were previously found for optimizing functions/program that are not similar at all with the new function/program. However, selecting solely based on a threshold can lead to situations where only a small number (or none) of the functions/programs from the reference set fall inside the similarity threshold. Nevertheless, as a fallback for this situation, a minimum number of the complete set of previously generated sequences may be used to generate the graph. In addition, in any of the approaches it could happen that a considerable number of the functions/programs selected are very similar between themselves, resulting in focusing exploration too much.

We believe it may prove itself important to weight the edges in the graph in a way that it reflects what functions/programs from the reference set are most similar with the new function/program, instead of considering them all equally. However, because of the potentially high dimensionality of the feature vectors (e.g., MILEPOST (Fursin et al., 2008) has 55 features), even very different functions end up with close cosine distances. Using the cosine values directly as a way to weight the importance of each sequence when building the graph would result in giving almost the same importance to the compiler sequences from all functions/programs in the reference set.

In order to give more consideration to sequences associated with functions/programs based on their cosine similarity with the new program/function, one can scale the similarity metric by applying a power of a experimentally determined number. Applying any positive power greater than 1, e.g., 100, results in further separating each pair of values for the cosine metric. For instance, $0.99^{100} = 0.366$ and $0.97^{100} = 0.048$. In this example, the influence of sequences previously found for the functions/programs in the reference set that have cosine similarity of 0.97 with the new function/program would be weighted $7.625\times$ less than functions with cosine similarity of 0.99. Larger powers (e.g., $1,000$) would result in a larger differences when weighting the impact of the sequences from the different functions/programs from the training set, and smaller powers (e.g., 10) would result in setting them less apart. This approach has the advantage of avoiding the problem of having to decide where to draw the line by considering the sequences associated with all functions/programs and rating each of them with higher or lower importance when building the graph. As to what would be the best power to use for weighting the contribution of the sequence (or sequences) associated with each function/program when generating the graph, that would be determined experimentally.

Rebuilding the graph only using the sequences associated with the most similar functions/programs in the reference set (or using sequences for all functions/codes but weighted differently) is functionally equivalent to using the full graph (i.e., built from sequences associated with all functions/programs in the reference set) and recalculating the probabilities of going trough each out-edge based on the next compiler passes and/or subsequences (and their frequency) in the sequences associated with the most similar functions/programs (or considering all with different

weights).

Instead (or in addition) of using static and/or dynamic features, an additional possible approach, and potentially more powerful, would use features in the IR generated after optimization to introduce a bias in the graph. An example of such approach would work as follows. Given a new function/program, and after selecting the first graph vertex, select the vertex based on similarities between the IR representing the new function/program after optimization with the first pass/subsequence represented by the first graph vertex selected and the IRs of the reference functions/programs after optimization with the same first pass/subsequence. The probability of selecting any of the next vertices reachable in the graph by out-edge connections from the current vertex would be recalculated based on the similarity between the transformed IRs corresponding to the reference functions/programs and the transformed IR of the new function/program. After selecting the second pass/subsequence to add to the sequence being generated, the selection of the third relies on similarities between the IRs resulting from application of the first two passes/subsequences in the IR of the new function/program and the IRs of the reference set; and so on for the selection of the next passes/subsequences to add during the generation of the candidate sequence.

In our experiments the DSE overhead is mostly caused by execution of the compiled functions. Overhead caused by compilation tends to be small in comparison. For targets where compilation time is more relevant, a different approach can be used. Instead of applying all the $n-1$ passes of a candidate sequence that is being built when selecting what pass/subsequence to add (i.e., what out-edge to select in the graph) to a candidate sequence, one can rely on a lookup table computed offline that has feature vectors extracted from all the versions of the IRs for the reference functions/programs after transformation by the subsequent application of the associated compiler sequences. For instance, for a given reference function/program associated with a given sequence A, for each pass/subsequence in A that is represented in the graph, there will be a feature vector for each IR generated from optimization of the base IR (unoptimized IR generated by the compiler frontend) with each pass/subsequence at start of sequence A. For any given vertex in the graph, during the process of generating a new sequence for a new function/program, the distances to the IRs of the reference functions/programs after application of that pass/subsequence would be computed. All the IRs (or feature vectors representing them) for the reference functions would be in a look up table, thus in this approach there would not exist additional overhead caused by applying compiler passes/subsequences to the reference functions/programs during exploration.

4.3 Targeting users that prioritize exploration efficiency

The overhead of exploring hundreds or even thousands of compiler sequences is a trade-of that some are willing to accept if they expect to be rewarded with considerable performance improvements (or improvements regarding other metrics). Conversely, this may not be the case for a considerable number of compiler users.

We experimented with a number of approaches as an attempt to find a way to achieve compiled code with high quality regarding an optimization metric in the context of DSE with a small number

of evaluations of compiler sequences (e.g., 10 or less). We experimented with an approach inspired by the work of Purini and Jain (2013). The key difference between their approach and other iterative approaches is that their method, given a new function/program, relies only on evaluating (up to) a comparatively small number of predetermined compiler sequences. One of their contributions was to show, using the Clang/LLVM compiler, that relevant binary execution performance improvements can be achieved relying on evaluating a small number of sequences representative of a function/program space of interest to the compiler user. This kind of approach presents an overhead much smaller than what is typical of iterative approaches that generate sequences online.

We experimented with different methods for the generation and selection of the compiler sequences to include in this set of sequences, converging to an approach that shares similarities with the work presented by Purini and Jain. On top of other differences, we experiment with a parameter that we experimentally found to result in improving binary execution performance of the compiled code and/or requiring less compiler sequence evaluations.

4.3.1 Particularities of the approach

In the experiments presented by Purini and Jain K sequences (e.g, $K = 10$) were selected from a list of 290 compiler sequences (366 before removing redundant sequences). This list of 290 compiler sequences was the result of selecting the best sequences, for each kernel in their reference set (61 kernels), for each of 6 DSE runs using 3 different iterative algorithms (2 variations per algorithm), including two GAs and uniform random search. Instead, in the experiments we present, we generate a much larger number of compiler sequences and extract the K sequences from the complete set of sequences generated/evaluated.

In the case of the experiments presented by Purini and Jain, the reduced set had 290 sequences ($6 * 61$ before removing redundant sequences). We use uniform random search to generate all sequences and we use a smaller number of reference functions (30) in our experiments. Considering 30 reference functions, any number of K sequences extracted from a set of only 30 sequences would not be able to cover many new functions/programs different from the the reference functions. The use of multiple DSE runs using different algorithms and/or exploration parameters in the experiments presented by Purini and Jain relieves this problem to some extent. Nevertheless, we opted to consider the complete list of sequences previously evaluated when extracting the K sequences, as there is a larger chance of covering the function/program space of the reference set and other sets of functions/programs.

Our approach relies on evaluating a considerably large number of sequences, incurring in a large initial overhead, instead of using methods that focus exploration when targeting a given kernel (e.g., GAs, SA), because we realized that having a K set composed of sequences that improve only a small number of functions (at the limit improving only a single kernel) would not work as well for compiling new functions/programs. This is evidenced by our tests with a new parameter that we introduce to the method we use for extracting the K sequences.

4.3.2 Extraction of highly representative sequences

The selection of the K compiler sequences from the initial set of sequences is performed by iteratively selecting sequences that result in improving more functions when combined with other sequences already selected. In case of a tie with other sequence(s), the sequence that results in the highest combined geometric mean speedup is selected.

The extraction approach we used is similar with the *Best-10* approach presented by Purini and Jain Purini and Jain (2013); although there is a key difference. The original method for extraction of the K sequences removes the columns of a table representing the improvement factors for functions/programs (represented as columns) from the reference set that are covered by a sequence (represented by a row in the table of improvements) selected to be part of the K sequences during the extraction process. If we did this, considering that we are extracting the sequences from a much larger initial set of sequences, we would end up loosing a considerable potential for improvement in both the reference set and new functions/programs. The much larger number of sequences to select the K sequences from, the fact that the reference set is composed of a limited number of functions, and the fact we use uniform random sampling to generate the initial set of sequences, results in the existence of a number of sequences in that set of sequences that improve all functions/programs from the reference set in comparison with the use of the compiler's standard optimization levels. Using the *Best-10* extraction approach as described by Purini and Jain results in removing all columns covering the reference set from the table representing improvements over baseline after the selection of the first sequence, thus leaving the table empty after the extraction of the first sequence.

While extracting each K sequence, the "combined" geometric mean speedup when selecting sequence number S is the geometric mean of the highest speedups for each reference function/program achieved by the use of the previously selected $S-1$ sequences, plus a candidate sequence being tested for selection.

4.3.3 Avoiding overspecializing to the reference set

We additionally experimented with a parameter that can improve performance on new functions/programs at the cost of reducing performance in the reference functions. The parameter adds the following requirement to any given sequence (from the initial set of sequences) for it to be a candidate sequence for the set of K sequences. Any sequence that if used to compile the functions/programs from the reference set results in a geometric mean speedup that is at a distance distance (given by a factor) larger that the value of that parameter in comparison the the highest geometric mean achieved with any single sequence from the set of all sequences evaluated (i.e., the sequence that results in highest geometric mean improvement) can not be considered for being part of the set of K sequences. We observed an effect similar with what is often called *overfitting* in the domain of machine learning. Adjusting this parameter, to which we call δ, can have the effect, when increased up to a certain value, of improving the overall effectiveness of the selected K sequences when compiling new functions/programs not included in the reference set.

4.3.4 Using the sequences to compile new programs/functions

Given a set of K sequences that has been extracted in order to cover a function/program space (e.g., one or more domains), and given a user that can only afford up to K compilations/executions, then their use is more likely to result in higher improvement of the quality of the function/program being compiled if these K sequences are used for compilation/evaluate in the order they were extracted. For instance, let's suppose the compiler user only affords to try 3 extra compiler sequences after evaluating –O3. With no additional information about any characteristics of the function/program being compiled, it will more likely be better to evaluate sequences 1, 2 and 3, over any other combination of sequences. As by the extraction methods used, sequences that are extracted first (i.e., sequences with lower index) are more likely to improve more functions from the reference set of functions, and if this set is representative, also more likely to improve new functions.

In order to make sense to evaluate by other order, characteristics about the functions/programs need to be taken into account. For instance, static or dynamic features of the functions/programs can be used to modify the order in which the K sequences are tested (e.g., evaluate sequence 4 before sequence 2). These features can be both features in source code or in the respective IRs before and/or after transformation. The combined factor of improvement over baseline of the sequences evaluated up to a point on a new function/program, could also be used as an input to the decision process that selects the next sequence to evaluate. For instance, for the reference set, it may be that for most functions if sequence 1 improves by X amount, sequence 2 improves by Y amount and sequence 3 improves by Z amount, then sequence 5 tends to be better than sequence 4, and therefore it may be better to evaluate it before. Note that evaluating the K sequences in a different order than the order they were extracted can result in severe penalties, as sequences that are extracted first will naturally tend to result in better coverage of a function/program space.

4.3.5 Advantages over other DSE approaches

This type of approach to the phase ordering problem has an advantage in terms of exploration time. By only compiling/evaluating at most with K sequences, while being able to achieve solutions better than the ones made possibly by the use of the –Ox compiler flags, it incurs in an overhead that can be accepted by larger number of users/programmers. Phase ordering is only performed offline for the generation of the initial table including the speedup for each code/sequence pair for a given target architecture, from which sequences are selected to form the K set.

The fact this type of approach relies on a fixed set of compiler sequences (i.e., determined offline), makes it possible for compiler writers to validate said compiler sequences using the same or similar methods they already rely on to validate the sequences represented by the –Ox flags. Validating compiler sequences generated by other enumeration based approaches (e.g., GAs, SA-based) is intrinsically more difficult, as traditional iterative DSE methods generate new previously unseen sequences. This can be a very important advantage, in comparison with traditional iterative approaches, specially in the case of using the sequences suggested by DSE in the process of compiling safe-critical applications. Unless the individual compiler passes are formally verified

there is no way to be completely sure for all types of functions/programs that the result of applying a given sequence generated by GAs, SA-based, or the graph-based approaches, does not result in breaking the compiler, or even worse, generating wrong code.

The set of K compiler sequences could be seen as an extended set of –Ox flags, where each domain or subdomain of programs/functions is at least particularly well covered by at least one of its sequences. The domain/subdomain can be in relation to anything that influences how the applications are written (e.g., industry sectors, programming style, optimizations already at source level), thus reflecting in the function/program features. The compiler user would then only have to evaluate an additional number of optimization levels when optimizing for his target architecture and optimization metric of interest.

Different sets of K sequences can exist specially tailored to specific domains, target metrics and target architectures. For instance, lets consider that a compiler is distributed with a set of K sequences for targeting performance, other set of K sequences for targeting energy consumption, and other set of sequences for reducing code size. Then, the user only has to select the target metric, and evaluate the new –Ox flags from the first to the Nth where N is the number of sequences he is willing to evaluate. Given that a number of compiler users are already familiar with the –Ox flags, evaluating additional optimization levels could be considered less cumbersome than exploring hundreds or thousands of sequences. However, performing a large number of evaluations would require the compiler user to use a DSE infrastructure such as ours. Instead, if the DSE process evaluating large numbers of sequences is done by the compiler developers or some other entity other than the final user, and only a set of K additional –Ox levels (per metric/target) are distributed with the compiler, then the compiler user does not need to know anything additional (e.g., would not need to deal with a DSE infrastructure) nor have more effort to better optimize functions/programs than when using the traditional optimization levels.

Additional sets of K sequences can be devised for targeting combined optimization metrics. For instance, a set of sequences K can exist to reduce code size as first priority but only if not more than 10% of the maximum performance is sacrificed. The only requirement is that a new set of K sequences needs to be extracted from the table with speedups achieved with sequences previously evaluated. If one already has data regarding the new metric(s) that one wants to consider for a new K set, then only a new extraction of K sequences needs to be performed. Otherwise, the functions/programs first need to be retested considering the new metric(s), which can also entail instrumenting the functions/programs for measuring the new metric(s).

4.3.6 Limitations of the approach

Comparing with other approaches (e.g., GAs, SA-based), this approach can be limited, in terms of the improvement over the –Ox optimization levels, by the particular sequences that are selected to be part of the K set. This will depend not only on the number considered for K, but also on the particular programs/functions used during the initial exploration phase for finding the K sequences, on the extraction method used, and on the DSE approaches used in the phase ordering stage of the approach.

For instance, if this initial set of functions/programs is limited to a single domain, compiling a new function/program from other domain with the K set of sequences may not achieve any improvement because of overspecializing to the reference set. But one could argue that in this case such situation can be seen as finding a set of K sequences that are best suited to a given domain, which also has its practical uses.

Nevertheless, even if the reference set of functions/programs that will determine the K set of sequences covers a very wide program space, GAs, SA, and the Graph-based DSE schemes, given enough iterations, are still expected to achieve the same or an even higher improvement of the quality of the generated code. This is not caused by only evaluating up to K sequences given a new function/program. In fact, K could be large (e.g., $K = 10,000$), giving the option to the compiler user to evaluate a much larger number of sequences. However, if many new –Ox levels are to be distributed with the compiler, it may as well come with a complete DSE infrastructure. What differentiates the GA-based, the SA-based and the graph approaches from this kind of approach is that they generate sequences online from a list of compiler passes, thus allowing for the generation of sequences that will never exist in a reasonably small set of K sequences. This allows further improving functions/programs that are not equally well covered by the K sequences.

4.4 Summary

We presented a SA-based approach that performs automatic specialization of the temperature parameters and the update factor.

We presented two approaches, IterGraph and SA+Graph, that use a graph representing favorable compiler subsequences to generate suitable optimizing compiler sequences faster. We also gave examples on how this graph-based approaches can be extended to take into account static and/or dynamic function/program features as a means to further prune the design space.

We presented an approach inspired in work by Purini and Jain (2013). This approach consists in using a reduced set of sequences (e.g., 10), extracted from experimental data generated with a set of reference functions/programs, to compile new unseen functions/programs. We use a parameter when extracting the reduced set of sequences to help improving the quality of the optimized code that is generated for new functions/programs from compilation with a given number of these sequences.

Additionally, we presented advantages and disadvantages of the approaches presented in relation to a number of aspects, such as exploration overhead and other particularities that can affect their potential to be favored by different types of compiler users.

Chapter 5

Experimental Results

This chapter includes the description of the kernels, the explanation on how our DSE system extracts a number of metrics for the target architectures, the description of the experiments and additionally, this chapter presents the experiments we performed as an effort to answer the research questions *Q1*, *Q2* and *Q3* in the context of binary execution performance.

Finally, this chapter presents experimental results achieved with the approaches presented in Chapter 4, addressing research questions *Q5*, *Q7* and *Q8*. The Simulated Annealing (SA)-based approach, the Graph-based approach and an approach inspired in Purini and Jain (2013) are compared in the context of different utilization scenarios: in a context where the user requires considerably faster exploration time.

See Appendix C and D for phase selection and ordering experiments targeting a NVIDIA Graphics Processing Unit (GPU) and reconfigurable hardware in the context of performance, measured as wall time and number of cycles, respectively. Appendix A presents phase ordering/selection experiments in the context of performance and energy consumption when targeting an Intel CPU from a workstation representative of a supercomputer node and an ARM-based ODROID-XU (Hardkernel) board. Information about how the metrics considered for these targets were extracted is presented in the respective appendices.

5.1 Experimental setup

This section presents the experimental setup used for the Design Space Exploration (DSE) experiments presented in the next sections. We present the program kernels used to evaluate our DSE schemes and we present details about the target platforms used. Our DSE system supports two different toolchains in the context of phase ordering. Specifically, a CoSy-based toolchain developed in the context of the REFLECT FP7 European project (REFLECT) and a Clang/LLVM based toolchain (the toolchains are described in Chapter 3). In both cases, we use LARA code inside a LARA DSE loop to generate compiler pass sequences, dictating what compiler passes to execute and in which order.

The results presented target different microprocessors, but we also performed experiments targeting other devices such as GPUs (See Appendix C) or reconfigurable hardware (see Appendix D).

The only requirement to use the LARA interpreter tool (i.e., *larai*) is the Java Runtime. Other requirements depend on the tools used by each particular LARA-aware toolchain used in conjunction with the LARA interpreter. The CoSy-based toolchain we use requires Ubuntu or CentOS Linux distributions, while the LLVM-based toolchain used supports a number of operating systems (e.g., Windows/MacOsX/Linux/BSD). Memory and processor requirements can vary, depending on the DSE algorithm used, the target platform, and other DSE parameters (e.g., number of DSE iterations).

5.1.1 Program kernels

We tested our approaches using kernels from distinct kernel sets. We rely on a set of kernels from Texas Instruments benchmarks (Texas Instruments, 2008a,b), a set of kernels from the REFLECT project (Cardoso et al., 2013b), a version of the PolyBench/C kernels (Pouchet et al.) with static memory allocation, and a set including a selection of kernels from other benchmarks.

REFLECT. This set contains two kernels, Filter Subband and Grid Iterate, respectively used by an MPEG decoder and a 3D path planning algorithm. These two kernels are provided by Coreworks and Honeywell, respectively, in the context of the REFLECT project (Cardoso et al., 2013b). The Filter Subband kernel is depicted in Figure 5.1. This kernel has two input arrays of type `float` (`z` and `m`) and output of the same type. A floating point version of the Grid Iterate kernel is depicted in Figure 5.2. It has a multidimensional array of `float` data type as input (the `obstacles` array) and an array of the same type and shape as output (the `potential` array).

Texas Instruments. The Texas Instruments set contains 42 kernels targeting embedded systems from two Texas Instruments C libraries, IMGLIB (DSP Image/Video Processing Library) (Texas Instruments, 2008a) and DSPLIB (DSP Signal Processing Library) (Texas Instruments, 2008b). Table 5.1 summarizes these kernels, including a description of each kernel and a check-mark for the kernels with a single execution flow (i.e., if they do not have multiple branches with conditional execution depending on the inputs).

PolyBench/C. We use the PolyBench/C (Pouchet et al.) benchmarks (version 4.1) in our experiments. The PolyBench/C benchmarks are floating-point data intensive and represent kernels from different domains, such as stencil computations, data mining, linear algebra and solvers. The PolyBench/C kernels used in the experiments presented in this chapter were generated using the included scripts to use the `MINI` input dataset and use static memory allocation. Table 5.2 enumerates the PolyBench kernels considered (all from PolyBench/C 4.1) in our experiments.

```
void filter_subband(float* y, float* s, float* z, float* m, int y_size, int s_size,
     int z_size) {
  int i,j;
  int j_max = z_size / y_size;

  for (i=0;i<y_size;i++)
  {
    y[i] = 0.0f;
    for (j=0; j<j_max;j++)
    {
      y[i] += z[i+y_size*j];
    }
  }
  for (i=0;i<s_size;i++) {
    s[i]=0.0f;
    for (j=0; j<y_size;j++) {
      s[i] += m[s_size*i+j] * y[j];
    }
  }
}
```

Figure 5.1: Filter Subband kernel.

```
void grid_iterate(int x_dim, int y_dim, int z_dim, int iter_steps_num, float
    potential[x_dim][y_dim][z_dim], int obstacles[x_dim][y_dim][z_dim])
{
  unsigned int i, j, k, steps;
  int val;
  float acc;
  float c = 1.0f/6.0f;

  for (steps = 0; steps < iter_steps_num; steps++) {
    for (i = 1; i < (x_dim - 1); i+=1) {
      for (j = 1; j < (y_dim - 1); j++) {
        for (k = 1; k < (z_dim - 1); k++) {
          val = obstacles[i][j][k];
          if (val == 1) {
            potential[i][j][k] = 0.0f;
          }
          else {
            if (val == -1) {
              potential[i][j][k] = 1.0f;
            }
            else {
              acc  =  potential[i-1][j][k] + potential[i+1][j][k] + potential[i][j
                  -1][k] + potential[i][j+1][k] + potential[i][j][k-1] + potential[
                  i][j][k+1];
              potential[i][j][k] = acc * c;
            }}}}}}}
```

Figure 5.2: Grid iterate kernel.

Table 5.1: Description of the Texas Instruments (TI) program kernels (Texas Instruments, 2008a,b).

Function	**Description**	**Single Execution Flow?**
DSP_autocor	Autocorrelation of an input vector.	✓
DSP_blk_eswap16	Endian-swap a block of 16-bit values.	✗
DSP_blk_eswap32	Endian-swap a block of 32-bit values.	✗
DSP_blk_eswap64	Endian-swap a block of 64-bit values	✗
DSP_blk_move	Move block of memory (overlapping). Endian Neutral.	✗
DSP_dotp_sqr	Dot product of two arrays.	✓
DSP_dotprod	Vector product of two input arrays.	✓
DSP_firplx	Complex FIR.	✓
DSP_firlms2	Least Mean Square Adaptive Filter.	✓
DSP_fltoq15	Convert the IEEE FP numbers into Q.15 format numbers.	✓
DSP_mat_mul	Matrix Multiply.	✓
DSP_mat_trans	Transposes a matrix of 16-bit values and user-determined dimensions.	✓
DSP_maxidx	Finds the largest element in an array (return index).	✗
DSP_maxval	Finds the maximum value of a vector (returns value).	✗
DSP_minerror	Minimum Energy Error Search,	✗
DSP_minval	Finds the minimum value of a vector (returns value).	✗
DSP_mul32	32-bit multiply, returning the upper 32 bits of the 64-bit result.	✓
DSP_neg32	32-bit vector negate.	✓
DSP_q15tofl	Q.15 to IEEE float conversion.	✓
DSP_vecsumsq	Sum of squares.	✓
DSP_w_vec	Weighted vector sum.	✓
IMG_boundary	Returns coordinates of boundary pixels.	✗
IMGonv_3x3	3x3convolution.	✓
IMGorr_gen	Generalized correlation with a 1 by M tap filter.	✓
IMG_dilate_bin	3x3 binary dilation.	✗
IMG_erode_bin	3x3 binary erosion.	✗
IMG_fdct_8x8	8x8 Block FDCT With Rounding, Endian Neutral.	✓
IMG_idct_8x8_12q4	IEEE-1180/1990 Compliant IDCT, Little Endian.	✓
IMG_mad_8x8	Minimum Absolute Difference motion search with 8x8 block	✗
IMG_median_3x3	3x3 median filter operation on 8-bit unsigned values.	✓
IMG_perimeter	Returns the boundary pixels of an image.	✗
IMG_pix_expand	Reads unsigned 8-bit values and store to a 16-bit array.	✓
IMG_pix_sat	Saturates 16 bit signed numbers to 8 bit unsigned numbers.	✗
IMG_quantize	Matrix Quantization w/Rounding, Little Endian.	✓
IMG_sad_8x8	Sum of Absolute Differences on single 8×8 block.	✓
IMG_sad_16x16	Sum of Absolute Differences on single 16×16 block.	✓
IMG_sobel	Sobel filter.	✗
IMG_wave_horz	1D Periodic Orthogonal Wavelet decomposition. This also performs a row decomposition in a 2D wavelet transform.	✓
IMG_wave_vert	Compute vertical wavelet transform.	✓
IMG_yc_demux_be16	De-interleave a 4:2:2 BIG ENDIAN video stream into three separate LITTLE ENDIAN 16-bit planes.	✓
IMG_yc_demux_le16	De-interleave a 4:2:2 LITTLE ENDIAN video stream into three separate BIG ENDIAN 16-bit planes.	✓
IMG_ycbcr422p_rgb565	Planarized YCbCr 4:2:2/4:2:0 to 16-bit RGB 5:6:5 conversion.	✓

Table 5.2: Description of the PolyBench/C 4.1 program kernels (Pouchet et al.).

Function	Description	Single Execution Flow?
2mm	2 Matrix Multiplications (alpha * A * B * C + beta * D)	✓
3mm	3 Matrix Multiplications ((A*B)*(C*D))	✓
adi	Alternating Direction Implicit solver	✓
atax	Matrix Transpose and Vector Multiplication	✓
bicg	BiCG Sub Kernel of BiCGStab Linear Solver	✓
cholesky	Cholesky Decomposition	✓
correlation	Correlation Computation	✓
covariance	Covariance Computation	✓
deriche	Edge detection filter	✓
doitgen	Multiresolution analysis kernel (MADNESS)	✓
durbin	Toeplitz system solver	✓
fdtd-2d	2-D Finite Different Time Domain Kernel	✓
floyd-warshall	Find shortest paths in a weighted graph	✓
gemm	Matrix-multiply C=alpha.A.B+beta.C	✓
gemver	Vector Multiplication and Matrix Addition	✓
gesummv	Scalar, Vector and Matrix Multiplication	✓
gramschmidt	Gram-Schmidt decomposition	✓
heat-3d	Heat equation over 3D data domain	✓
jacobi-1d	1-D Jacobi stencil computation	✓
jacobi-2d	2-D Jacobi stencil computation	✓
lu	LU decomposition	✓
ludcmp	LU decomposition followed by Forward Substitution	✓
mvt	Matrix Vector Product and Transpose	✓
nussinov	Dynamic programming algorithm for sequence alignment	✗
seidel-2d	2-D Seidel stencil computation	✓
symm	Symmetric matrix-multiply	✓
syr2k	Symmetric rank-2k operations	✓
syrk	Symmetric rank-k operations	✓
trisolv	Triangular solver	✓
trmm	Triangular matrix-multiply	✓

5.1.2 Bare-metal targets

We use our phase selection/ordering exploration system to target different systems. The following subsections present the bare-metal platforms we targeted in the experiments presented in this chapter. For each target, we had to add a performance reporting component to our DSE system, in order to be able to present performance improvements obtained with phase ordering.

In addition, our DSE system supports optimizing for code size when targeting all supported architectures (see, section 3.8). A number of embedded systems have Read-Only Memory (ROM) size limitations, often totaling just a few KBytes. To measure code size we use the value reported by the *size* utility. It reports the memory space taken by function/program instructions (*text* field), the space taken by initialized variables (i.e., variables that live in ROM, *data* field), and the space taken by non-initialized variables. The metric extractor of our DSE system reports the sum of the *text* and *data* fields as the code size of a given object code compiled from an input function with a given compiler sequence.

We present the four different microprocessors/microcontrollers microarchitectures that we targeted in context of bare-metal execution, each selected because of its relevance, ease of access to cycle accurate simulators, or both. Given that the execution is bare-metal (i.e., no operating system scheduling threads), we can measure the number of execution cycles reliably with a single simulation/execution using cycle accurate simulation/execution environments. Table 5.3 depicts general information about the targets.

Table 5.3: General information about the considered Microprocessor/Microcontroller cores.

Core	**Owner**	**ISA**	**Available in?**	**License**	**Soft/Hard core?**
MicroBlaze (Xilinx)	Xilinx	MicroBlaze	Netlist	Requires Xilinx EDK license	Soft core for Xilinx FPGAs
LEON3 (Cobham Gaisler AB, a)	Aeroflex Gaisler	SPARC V8 w/ V8e	VHDL	GPL or private commercial	Soft/Hard core
Cortex-M4 (ARM)	ARM	ARMv7E-M (M4)	Silicon	Commercial	Hard core
Mor1kx (Baxter and Kristiansson)	GitHub project	OpenRISC 1000	Verilog	Open Hardware Description License	Soft core

All processors/microcontrollers targeted are supported by the GNU toolchain and by LLVM, either officially or unofficially (i.e., not on the main LLVM builds). In the experiments presented in this thesis we targeted MicroBlaze (Xilinx) using the CoSy-based REFLECTC compiler and

the Clang/LLVM compiler toolchain version 3.3, as it is the last version with support to the MicroBlaze target. The LEON3 (Cobham Gaisler AB, a) and the ARM Cortex-M4 (ARM) continued being supported by the official LLVM build, so we use the latest release of Clang/LLVM available when performing our experiments. OpenRISC is not currently supported by the official LLVM build, therefore for the Mor1kx (Baxter and Kristiansson) target platforms, we use Clang-OR1K [1] and LLVM-OR1K [2], which are based on up to date versions of Clang and LLVM.

Table 5.4 depicts more details about the microarchitecture of the target microprocessors/microcontrollers. All use the Harvard memory architecture, i.e., they have separated data and instruction memories. The ARM Cortex-M4, one of the most powerful microcontrollers, is the only hardcore (i.e., processor with physical implementation) used in our experiments. All other targets are simulated.

Table 5.4: Architecture details of target microprocessors/microcontrollers.

Core	**Pipeline**	**Cache**	**Opt Instructions**	**FPU**
MicroBlaze	3- or 5- stage	Optional and configurable	Multiply, divide, and floating-point.	Optional
LEON3	7-stage	Optional Random, LRR or LRU replacement.	Non-applicable	Interface with custom FPUs Fully pipelined IEEE-754 (LEON3, LEON4)
Cortex-M4	3-stage with branch speculation	No	Non-applicable	Optional Single-precision FPU (M4)
Mor1kx	6-stage (cappuccino) or 2-stage (espresso)	Each cache has a default size of 8 KB, but both are individually scalable between 1 and 64 KB.	ORFP32X ISA implementing IEEE-754 compliant single precision floating	Optional

Compilation is performed with the LLVM static compiler using the `-march=sparc` and `-mcpu=v8` flags for LEON3, `-march=thumb` and `-mcpu=cortex-m4` for the ARM Cortex-M4, `-march=or1k -mcpu=or1200` for OpenRISC targets, and `-march=mblaze` for MicroBlaze targets, with `-mcpu=mblaze3` or `-mcpu=mblaze5` depending on the MicroBlaze version.

[1] Available at https://github.com/skristiansson/clang-or1k, last accessed 21 July 2017.
[2] Available at https://github.com/openrisc/llvm-or1k, last accessed 21 July 2017.

LEON3. The LEON3 (Cobham Gaisler AB, a) is a softcore free to use for non-commercial purposes that is being extensively used in Space projects. The LEON3 was simulated using Aeroflex Gaisler's TSIM2 cycle accurate simulator (version 2.0.46) (Cobham Gaisler AB, b, 2016). TSIM2 supports non-intrusive execution time profiling for ERC32 and LEON2/3/4 processors. TSIM2 reports execution time as the number of cycles required to execute a given input program, along with profiling information about the percentage of execution cycles spent on each function. TSIM2 has other interesting features, such as accelerated processor standby mode, Memory Management Unit (MMU) emulation and SRAM emulation with cycle accurate timings. Simulation performance is said to be up to 60 MIPS on a Intel i7-2600K@3.4GHz (Cobham Gaisler AB, 2016). We use the default configurations, which simulates a LEON3 core with 4,096 KB of SRAM memory in a single bank, 32 MB of SDRAM memory in a single bank, 2,048 KB of ROM memory, and 4 KB each of instruction cache and data cache (16 bytes per line in both caches). In order that the exploration infrastructure is able to access information about the number of clock cycles required to execute a given function optimized with a given phase order, we passed the following list of parameters to TSIM2: `prof 1 1`, `run`, `perf`, `prof`, `reg`, `quit`. This particular order of parameters instructs the simulator to enable profiling with 1 clock cycle as sampling period, resets the simulator and starts execution from address 0, prints execution statistics, and prints the Integer Unit registers. The sample period for profiling is by default 1,000 cycles to provide higher simulation performance. This would not be acceptable for some of the faster kernels we use in our experiments, for which optimized versions can execute in less than 1,000 clock cycles. Given this fact, in order for us to confidently refer that a given optimized version is 1% faster than another version, we need a period smaller than 10 clock cycles, as 1% of 1,000 is 10. In our experiments targeting LEON3 we set the sampling period to 1 in order to obtain the maximum possible accuracy.

ARM Cortex-M4. ARM CPUs/microcontrollers are very populars in a number of domains. We use the STM32F411RE Nucleo development board (STMicroelectronics), which has an ARM Cortex-M4 core with DSP and FPU, 512 Kbytes of Flash memory, a 100 MHz CPU, and adaptive real-time acceleration allowing 0-wait state execution from Flash memory. In order to prepare the code to compile for the ARM Cortex-M4 we relied on the STM32Cube (version 1.3.0) embedded software libraries targeting that specific microcontroller. We use the microcoontroller ASM (*startup_stm32f411xe.s*) and C initialization code (for CPU and peripherals) from the UART project example targeting the STM32F411RE Nucleo development board. All the required steps regarding the transfer of the executable codes to RAM or Flash memory are performed automatically by the backend targeting this board that we developed for our DSE system. These steps are executed in a loop when evaluating each new compiler pass phase selection/ordering. When C source code is passed as input to our DSE system, the initialization and instrumentation code is automatically added before compilation. Executable files are transferred between the computer responsible for the compilation and the board connected to it through USB using the ST-LINK/V2 in-circuit debugger and programmer. On the software side, Open On-Chip Debugger (version 0.9.0) and GDB from Sourcery CodeBench Lite Edition for the ARM ELF target (version 2014.05)

are used. GDB interfaces with STLINK, through Open On-Chip Debugger to reset and halt the microcontroller, load a given ELF file, restart the microcontroller, and stop execution when execution of the *main()* function terminates (detected by the `bx lr` assembly instruction after `bl main`).

Mor1kx. OpenRISC cores are used in research and in some commercial products. We use the Mor1kx open source OpenRISC 1000 core, because it has an up to date Verilog implementation. The OpenRISC 1000 processor was simulated using Verilator (version 3.856) (VeriPool), a free Verilog simulator which compiles synthesizable Verilog into C++ or SystemC code. The Mor1kx OpenRISC processor was alternatively used in one of the two options: an area optimized version with a 2-stage pipeline, and a performance optimized version with a 6-stage pipeline, also known as Mor1kx Espresso and Mor1kx Cappuccino, respectively. The Mor1kx Cappuccino has a branch predictor unit, while the area optimized versions, the Mor1kx Espresso and the Mor1kx Pronto Espresso (another variant of the Mor1kx), do not. The Mor1kx core was configured using the default configuration of the *mor1kx-generic* system from the *orpsoc-cores* project [3], with the `OPTION_CPU0` (rtl/verilog/orpsoc_top.v) set to `CAPPUCCINO` or `ESPRESSO`. Our Verilator testbenches for the OpenRISC CPUs are developed on top of the testbenches from *orpsoc-cores*. The OpenRISC testbenches from the *orpsoc-cores* project do not allow to directly measure the number of cycles taken to execute a given function. They only report the number of clock cycles resulting from executing the complete program, i.e., the value reported also reflects initialization code and other calls in `main()` or calls to other functions. In order to extract the number of cycles for execution of the optimized functions using Verilator, we use the no operation (NOP) assembly instructions. The encoding of the NOP instructions allows an immediate to be passed. We developed the mechanism to count the number of cycles for the execution of the function being optimized, and added it to the Verilator testbenches. The only modifications needed in the source code to use this mechanism to report clock cycles is to instrument the call site of the function to optimize as shown in Figure 5.3. This is performed automatically by the DSE system. We adapted the testbench executing Verilator to start and stop counting cycles when NOP instructions with literals $0x000d$ and $0x000e$ are reached, respectively. The implementation of this mechanism allows for very precise measurements, given the fact that simulation is done at RTL level using a Verilog description of the Mor1kx cores, and we are measuring only the execution time of the functions being optimized by DSE of compiler phase orders.

MicroBlaze. The MicroBlaze processor (Xilinx) is the most used softcore in projects using Xilinx FPGAs. The two versions of the MicroBlaze used in our experiments, an area optimized 3-stage version and a performance optimized 5-stage version, are simulated using the ACE simulator. Both versions of the MicroBlaze used in our experiments include a barrel shifter, a divisor and a multiplier. None of the MicroBlaze configuration we use includes a FPU. Compared with

[3] Available at https://github.com/openrisc/orpsoc-cores, last accessed 12 January 2017.

```
int main() {
  ...
    asm("l.nop 13");
    function(...);
    asm("l.nop 14");
    ...
}
```

Figure 5.3: Generic fine-grained performance profiling in OPENRISC target using NOP instructions.

the simulators used for the other targets, the simulator used for the MicroBlaze targets has the limitation of only reporting clock cycles for the whole execution of a program. This has the effect of reducing the speedups calculated from the number of clock cycles reported, thus the performance improvements for the function being optimized are larger than reported.

5.2 Phase ordering in different CPUs

We present in this section experiments we performed early during the Ph.D. as a first effort to address some of the research questions introduced in Chapter 1. More specifically, research questions *Q1* and *Q2* in the context of execution performance of a set of functions/programs, when targeted to a number of different platforms.

Our initial effort to achieve the goal of understanding how much can functions/programs improve by the use of specialized compiler sequences and how tied are these sequences to a given target architecture consisted in finding sequences for a number of different CPU targets using our DSE scheme based on SA and evaluating the sequences found to result in compiled code with highest performance when executed on each of the six CPUs considered on the other CPUs.

The experiments were performed on the following microprocessors/microcontrollers: an ARM Cortex M4, a LEON3, an area-optimized OpenRISC Mor1kx (2-stage), a performance optimized OpenRISC Mor1Kx (6-stage), an area-optimized MicroBlaze (3-stage) and a performance-optimized MicroBlaze (5-stage). Iterative compilation was performed considering a set of 39 Texas Instruments kernels, relying on one of the earliest implementations of our DSE system for exploring compiler phase orders for the LLVM compiler.

For the experiments presented here, the SA parameters are set in a way that the algorithm allows the first iteration a 50% chance of selecting a compiler sequence that results in 100% worse performance than the previously selected, and the algorithm gives a 1% chance of selecting a sequence that results in 0.001% worse performance at the last iteration. The minimum temperature (T_{min}), the maximum temperature (T_{max}) and the cooling factor (α) are automatically specialized accounting for the kernel and target platform, as described in Chapter 4. The SA-based DSE scheme was executed for $10,000$ iterations, while considering sequences of up 32 LLVM compiler passes, considering only the passes in the –O3 flag.

We used LLVM 3.5 when dealing with all microprocessors except the ones with the MicroBlaze ISA, that require using an older version of LLVM (we used version 3.3). Passes are selected by the SA-based DSE scheme from the set of all passes that are present in the –O3 sequence of the particular LLVM compiler version used. Sequences can include multiple instances of the same pass without any constrain in relation to their position.

5.2.1 Multi-target SA-based LLVM compiler sequences exploration

Performance improvements of up to 2× over the best LLVM –Ox, i.e., compared with –O1, –O2, and –O3, depending on which results in compiled code with higher performance (per kernel), were achieved for 6 kernel/target pairs, as can be seen in Figure 5.4. Our SA-based DSE scheme found sequences that result in better performance than the best experimentally found –Ox, for between 32 and 36 kernels out of a total of 39 kernels, depending on the microprocessor targeted. Interestingly, the particular kernel codes that resulted in a higher speedup in comparison with the default optimization levels were not the same across the different target processors.

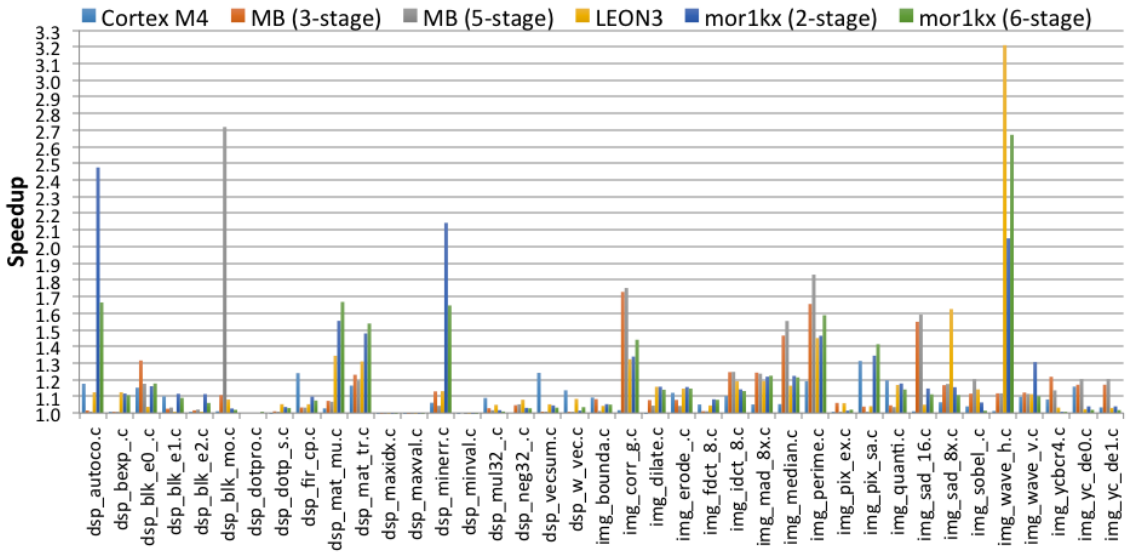

Figure 5.4: Speedups obtained after 1,000 iterations of the SA-based DSE scheme for each target/kernel pair when compared with the performance of the –Ox sequence resulting in highest performance for the same target/kernel pair.

Figure 5.5 presents, for each of the microprocessors considered, different intervals of factors of improvement achieved over compilation with LLVM –Ox, and for each interval, the number of kernels that were improved by a factor falling in that interval. Considering all microprocessor targets, the code generated for most kernels improve up to 1.1×. The second most common performance improvement factors fall between 1.1× and 1.2×, and the third most common between 1.2× and 1.3×. The remaining performance improvements are distributed between 1.3× and 1.7× and, in a smaller number, between 1.7× and 3.3×.

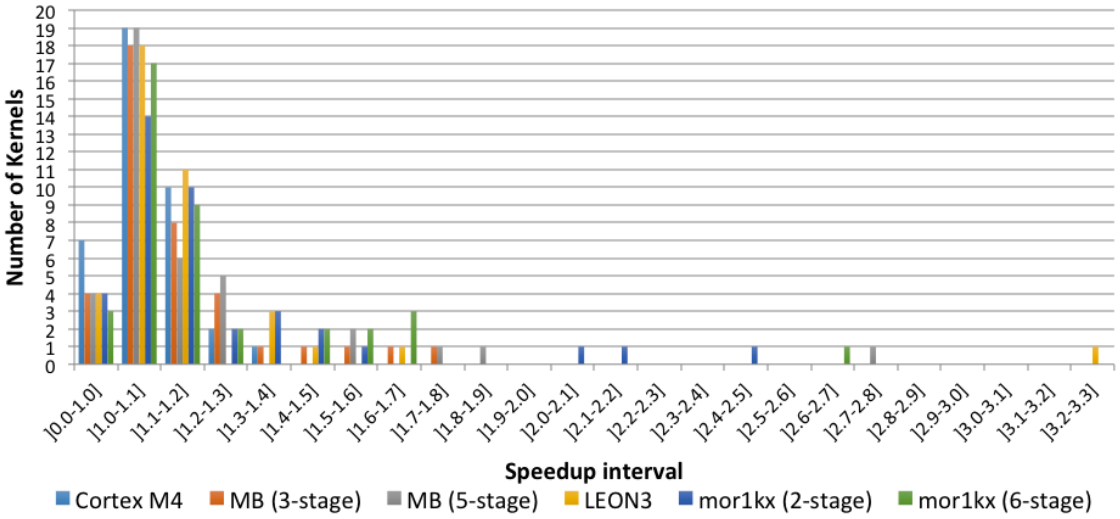

Figure 5.5: Histogram that represents the number of kernels for which it was possible to achieve a speedup that falls in each interval.

Figure 5.6 shows, as an example, a number of kernels for which DSE was unable to find better sequences than the sequences used when relying on the –Ox flags, at least when targeting one of the microprocessors considered. The number of such cases was 7 for Cortex M4, 3 for the 6-stage mor1kx, and 4 for all the other targets.

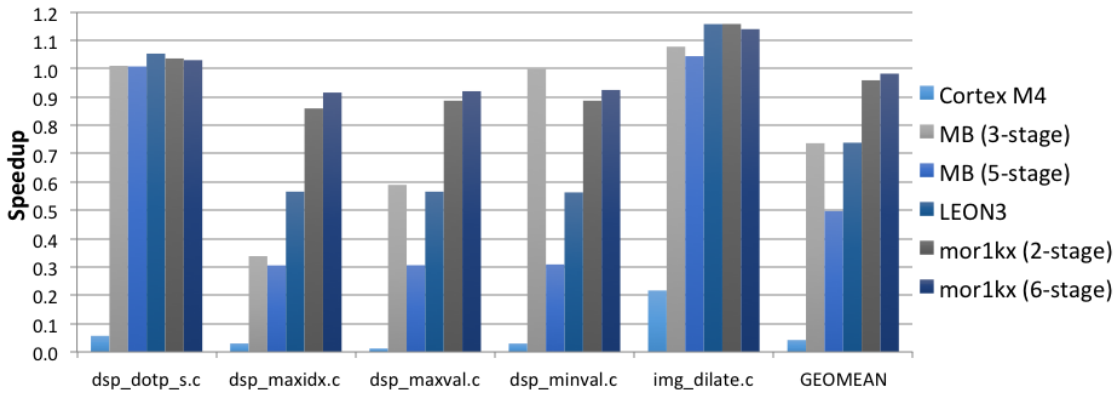

Figure 5.6: Program kernels where the performance of the best sequence found for some target(s) is worse than the performance resulting from compilation with the best individually selected –Ox.

Figure 5.7 presents two geometric means (*Geomean 1* and *Geomean 2*). The former (*Geomean 1*) is the geometric mean of the performance factors achieved for all kernels when targeting each microprocessor, including the slowdowns. The latter (*Geomean 2*) considers the likely scenario where the user evaluates the –Ox flags, and uses the best –Ox flag instead of the compiler sequence found by DSE if it ends up being worse. For instance, the *Geomean 1* speedup for the Cortex-M4 is 0.72×, i.e., there is a slowdown in performance of the compiled code when considering

for evaluation only the sequences found by DSE for this target. On the other side, the *Geomean 2* represents a performance improvement of 1.08×. We consider *Geomean 2* to give a better representation of the general performance improvements that can be expected to be achieved with exploration in a realistic scenario. Overall we are able to find compiler sequences that result in geomean performance speedups between 1.08× (for the Cortex-M4) and 1.20× (for the 6-stage Mor1kx), just by reordering and activating/deactivating compilation stages from the –Ox sequences.

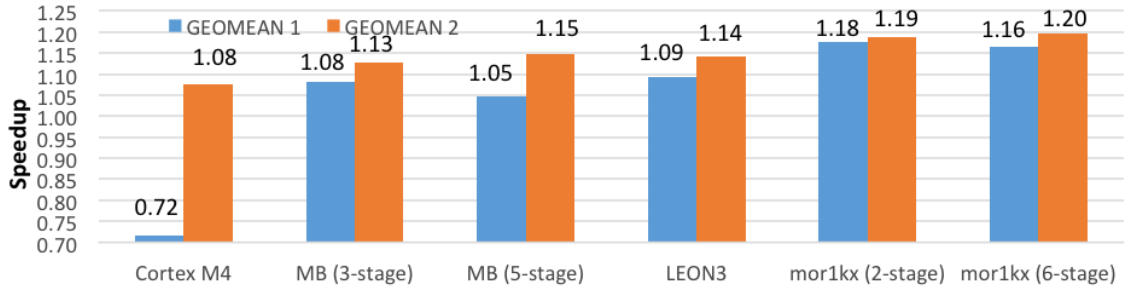

Figure 5.7: Geometric mean of speedups of best found sequences (GEOMEAN 1) and of using the best –Ox in case no better sequence is found (GEOMEAN 2).

5.2.2 Exploration time for all target/kernel pairs

Figure 5.8 depicts the exploration time using an early version of our SA-based DSE approach. The exploration time is of the same order of magnitude for all targets except for OpenRISC (i.e., the Mor1kx) and ARM (i.e., the Cortex-M4). The former is simulated using a Verilog simulator which results in comparatively slower simulation when compared with simulators specially tailored to specific microprocessors/microcontrollers (e.g., TSIM2); and the latter is executed on a development board, which although executes the compiled code quite fast, is limited by a slow connection (9.7 Kbytes/s in low speed, 12.8 Kbytes/s in high speed) provided by the STMicro's STLink 2.1 in-circuit debugger that serves as interface between a host computer and the evaluation board with the ARM Cortex-M4 microcontroller that was used in our experiments.

Notice that these DSE execution times represent exploration without the technique described in Chapter 3 that reduce DSE overhead by not evaluating binaries previously evaluated. That feature was not yet part of our DSE system at the time these experiments were performed. So, these execution times are in fact representative of the time taken to compile and execute/simulate $10,000$ times.

Figure 5.9 presents results for a scenario where the compiler sequences found for a given set of kernels when targeting an architecture are used to target the same kernels to other architecture. The MicroBlaze and LEON3 processors are each targeted using the compiler sequences found by our system to the other microprocessor (i.e., MicroBlaze sequences → LEON3 processor and LEON3 sequences → MicroBlaze processor).

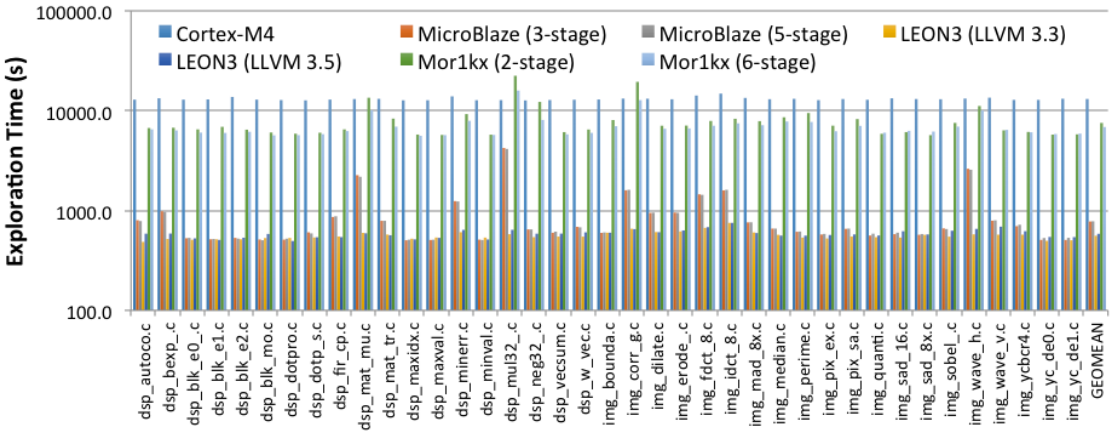

Figure 5.8: DSE time for all LLVM targets/kernels (10,000 iterations).

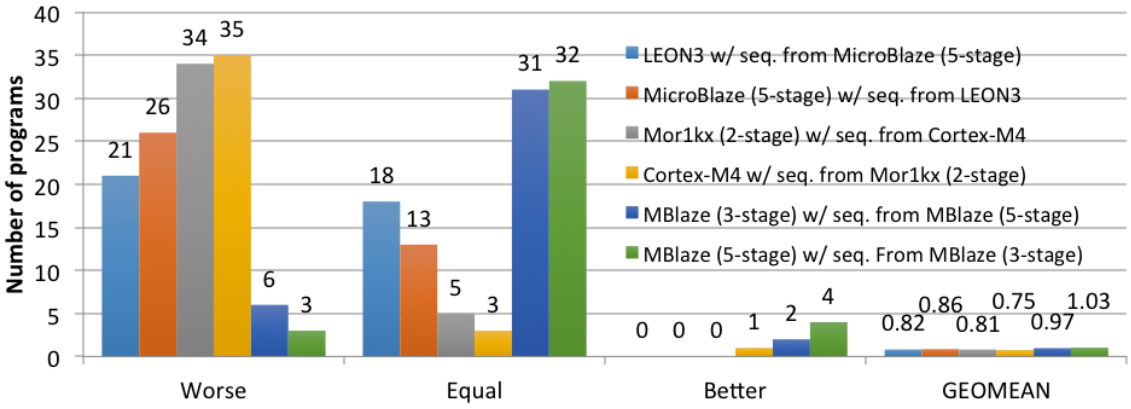

Figure 5.9: Histogram representing number of kernels from the set of Texas Instruments kernels for which worse, equal or better performance is attained when using compiler sequences found for other target.

The performance suffers a hit, both when targeting the MicroBlaze and the LEON3 using the compiler sequences found for the other microprocessor. In two kernels, the performance was below 10% (`img_fdct_8.c` and `img_idct_8x.c` with LEON3 processor and MicroBlaze sequences) of what was achieved with the compiler sequences found for the microprocessor. In this scenario, the geometric mean speedups are 0.816× and 0.861× (i.e., slowdowns), respectively for the LEON3 and the MicroBlaze processors. This is not surprising as the MicroBlaze and the LEON3 are quite different. There is a performance degradation by using compiler sequences found for a target microprocessor when targeting other microprocessor in most cases, except for the `dsp_mat_mu.c` and `dsp_mat_tr.c` kernels when targeting the Mor1kx processor using the sequences found for the Cortex-M4, and for the `dsp_neg_32.c` and `img_pix_ex.c` when targeting the Cortex-M4 with the sequence found for the Mor1kx.

As expected, the performance when using the best sequences found for the 5-stage MicroBlaze on the 3-stage MicroBlaze, and vice-versa, neither improves nor decreases for most kernels. On an example (`img_bounda.c`) the performance decreases in both situations. On the same 6 kernels the performance improves for one target (i.e., MicroBlaze 5-stage or 3-stage) and decreases for the other (`dsp_blk_e0.c`, `dsp_blk_mo.c`, `dsp_mat_tr.c`, `dsp_maxval.c`, `dsp_minval.c`, and `img_sobel.c`), and for 1 kernel the performance decreases only for one of the MicroBlaze processors (`img_corr_g.c`).

5.2.3 Using sequences found for a given version of LLVM with other version of LLVM

Figure 5.10 depicts the results achieved using the compiler sequences found for LLVM 3.3 on LLVM 3.5 and the results for the inverse situation, in both cases in the context of compilation targeting the LEON3.

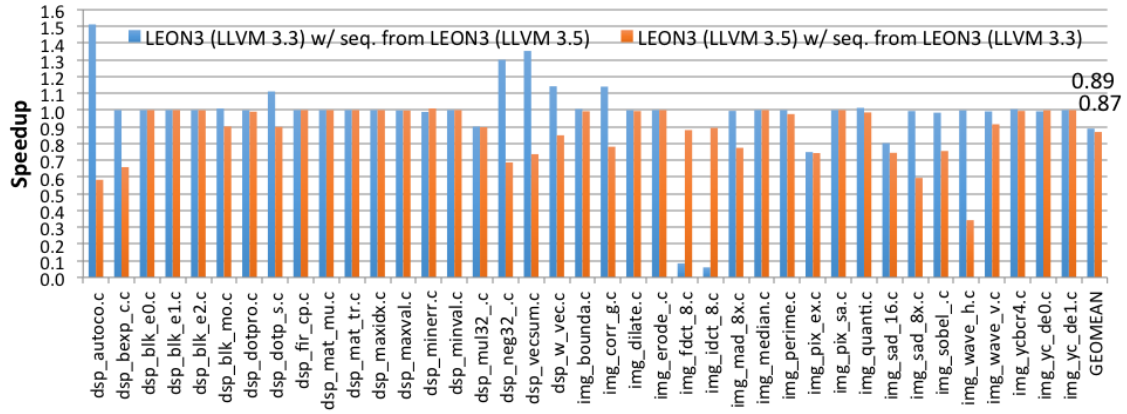

Figure 5.10: Targeting a LEON 3 with LLVM 3.3 using sequences found for LLVM 3.5 and with LLVM 3.5 using sequences found for LLVM 3.3.

When using compiler sequences found for the LEON3 core using LLVM 3.3 on LLVM 3.5, the performance of the compiled code (i.e., the SPARC v8 assembly) was consistently equal or

worse, with exception of `dsp_minerr.c`. For the inverse situation, i.e., using sequences found for LEON3 using LLVM 3.5 on LLVM 3.3, there were eight kernels that resulted in better performance. This is the result of LLVM 3.5 using compiler passes as part of the –Ox compiler sequences that are not present in the –Ox compiler sequences in LLVM 3.3. LLVM 3.3 –O3 has 48 unique passes (78 instances), while LLVM 3.5 –O3 has 53 unique passes (96 instances). The following passes exist in LLVM 3.5 –O3 but not in LLVM 3.3 –O3: *-verify-di*, *-mldst-motion*, *-slp-vectorizer*, *-barrier*, *-branch-prob*, *-block-freq* and *-loop-vectorize*. Passes *-preverify* and *-simplify-libcalls* exist in LLVM 3.3 –O3 but not in LLVM 3.5 –O3.

The performance ratios when using compiler sequences found for other compiler version were 0.889× and 0.869× for LLVM 3.3 and LLVM 3.5, respectively. This caused us to wonder about what kinds of possible performance improvements were still untapped. Maybe future versions of LLVM will include in their –Ox sequences, compiler passes that although are not yet part of the –Ox sequences used in the most up to date stable LLVM release (v. 3.5), they are still available to use; in the same way the loop vectorization compiler pass was already available in LLVM 3.3, although not present in any –Ox sequence at the time.

5.3 Experiments with the graph-based approach

We tested the graph-based approach targeting a LEON3 (Cobham Gaisler AB, a) core with LLVM 3.7, using the LLVM Optimizer (to optimize LLVM IR generated by the Clang frontend) and the SPARC backend (with flag for V8 architecture) of the LLVM Static compiler. The selection of the compiler passes and their execution order with the LLVM framework is accomplished by passing the flags representing the passes in a specific order to the LLVM Optimizer tool (a.k.a. opt). After the LLVM IR is transformed by the execution of the compiler passes in the requested order, the DSE system calls the LLVM static compiler (a.k.a. llc), to generate assembly code and then linked with a GNU Linker from the LEON Bare-C Cross Compilation System.

Specialization of compiler phase orders is performed at function level. Each function to optimize goes through a separated compilation flow, which includes the search for and the compilation with a specialized compiler sequence.

We use a set of programs/functions targeting embedded systems from two Texas Instruments C libraries, DSPLIB (DSP Signal Processing Library) (Texas Instruments, 2008a) and IMGLIB (DSP Image/Video Processing Library) (Texas Instruments, 2008b); and a set of 11 C functions (*adpcm_code*, *adpcm_deco*, *autcor*, *bubble_sort*, *dotprod*, *fdct*, *fibonacci*, *max*, *min*, *popcnt*, *sobel*) of the embedded systems domain used as reference for the clustering approach presented in (Martins et al., 2014b,a, 2016).

For the experiments presented in this section, the compiler pass graph instances are built from compiler sequences found using a SA-based exploration process, considering all LLVM optimizer (a.k.a. opt) compiler passes except the ones related to visualization (e.g., *view-**), printing (e.g., *print-**), generating dot graphs (*dot-**), and compiler passes that require loading external information (e.g., *insert-gcov-profiling*) or the use of external modules (e.g., *asan*, *tsan*). Note, however,

that we could have used sequences found using other iterative algorithms (e.g., a Genetic Algorithm).

We validated our approach with two experiments. First, we performed leave-one-out cross-validation using only the Texas Instruments functions; i.e., 42 different instances of the graph were created using 41 compiler sequences each (all sequences except the one for the function being compiled). We used the same graph instances both in our graph-based approach and as the structure to guide an implementation of the SA-based algorithm described in Section 4.2.7. Second, and last, we used our DSE method to search for compiler sequences for the same Texas Instruments functions using a single graph instance built using sequences previously found for the set of 11 additional functions.

For the leave-one-out experiments, the number of nodes of the graph was 140 and the average number of edges was 2,353 (minimum: 2,280, maximum: 2,402). For the second set of experiments, the graph has 98 nodes and 305 edges.

We compared the graph-based approach with other algorithms; which we implemented in our DSE infrastructure. The algorithms we considered for the experiments, in order to compare with the graph-based approach, include a sequential insertion based algorithm (Huang et al., 2013), a GA-based algorithm (Martins et al., 2014b,a, 2016), the SA-based algorithm (Nobre, 2013), and the SA+Graph (Nobre et al., 2015) approach that uses the graph to guide our SA approach.

5.3.1 DSE system

We rely in our in-house modular DSE infrastructure (see Chapter 3). Exploration schemes, objective functions, and target-specific exploration and configuration parameters are programmed with the LARA aspect-oriented programming language (Cardoso et al., 2012a). Modularity is assured by the fact that a new DSE infrastructure component (e.g., new objective function) can be paired with other components from other types (e.g., available targets and available exploration schemes).

Instead of relying on code annotations, code transformations and compiler mapping strategies are described in a separated file; allowing the reuse of theses transformation/mapping specifications across different sources/applications and/or targeting multiple platforms/requirements using a single annotation-free source code.

The DSE algorithms are programmed in a completely separated way and are independent from objective functions, from the compiler toolchain used and from the simulator calls or hardware interfacing (e.g., sending code and getting result from a board through special link or TTL 232R) specific to the target platform. The framework provides means to abstract calls to the individual tools of a given toolchain (e.g., LLVM Optimizer, LLVM Static compiler) from inside the LARA execution environment with a given set of parameters (e.g., compiler flags). We have developed an interface between the LARA framework and LLVM, which allows calling the Clang frontend, the LLVM Optimizer and the LLVM static compiler from LARA aspects. Relying on a simple interface, a LARA programmer can access the Clang/LLVM toolchain and instruct the compilation of any given source code considering the execution of a sequence of compiler passes at an arbitrary order.

The code generated is verified by representative tests (see Section 3.5). Developers are open to use optimized versions using non-standard compiler sequences knowing that they were not possibly previously tested or they were tested to a limited extent.

5.3.2 DSE parameters

When comparing the algorithms, we considered the execution of each DSE algorithm with the parameters previously explained for different numbers of iterations. More specifically, 10, 100, 1,000, 10,000 and 100,000 compilations/simulations. In the case of Sequential Insertion and the GA-based method, we neither consider the execution for 10 nor for 100 iterations, as they represent a too low number of iterations for those algorithms. They are neither enough iterations for Sequential Insertion to consider the insertion of all compiler passes, nor enough iterations for the GA-based considering as we have chosen the initial population to be composed of 300 compiler sequences. In the case of the Sequential Insertion algorithm, the number of iterations does not represent the total, but instead, the maximum number of iterations; as one of the stopping conditions of the algorithm considers all compiler passes for insertion in all positions of the candidate sequence, without any insertion.

A reduction in exploration time is expected at the expense of disregarding some compiler sequences, which may in some cases result in not allowing the DSE algorithm to achieve better solutions. One of our goals is to measure how much exploration time can be reduced without sacrificing too much the quality of the solutions.

The maximum sequence length was set to 128; i.e., the generated sequences are composed of up to 128 compiler passes. This length was experimentally found to not limit the potential for speedup, while reducing the number of sequences that resulted in problems. We select a maximum of 128 compiler pass instances (compiler passes can appear multiple times in the same sequence) from the graph, which has 140 nodes, corresponding to the LLVM passes represented in Table 5.5.

By default, the target of our exploration processes is performance. Therefore the objective function passed to the DSE methods consists on minimizing the number of CPU clock cycles needed to execute a particular function.

The GA-based, the SA-based, the SA+Graph and the Graph-based exploration algorithms are stochastic methods. For the experiments presented in this section, we executed the algorithm with the same number of iterations for three times and registered the geometric mean of the performance speedups achieved with each individual execution.

For the experiments presented here, and considering the Graph-based approach, the first node visited in the graph is the one corresponding to the compiler pass more frequently present in the compiler sequences used to generate the graph, i.e., the *loop-rotate* compiler pass. We found out experimentally that this is a good heuristic for determining the first pass of large sequences.

Table 5.5: LLVM Optimizer compiler passes used for exploration.

-aa-eval	-deadargelim	-ipsccp	-lowerbitsets	-rewrite-symbols
-adce	-debug-aa	-irce	-lowerinvoke	-s.-c.-o.-f.-gep
-add-dis.	-delinearize	-iv-users	-lowerswitch	-safe-stack
-alig.-f.-ass.	-die	-jump-threading	-mem2reg	-sancov
-alloca-hoisting	-divergence	-lazy-value-info	-memcpyopt	-scalar-evolution
-always-inline	-domfrontier	-lcssa	-memdep	-scalarizer
-argpromotion	-domtree	-libcall-aa	-mergefunc	-scalarrepl
-ass.-cache-track.	-dse	-licm	-mergereturn	-scalarrepl-ssa
-atomic-expand	-early-cse	-lint	-mldst-motion	-sccp
-barrier	-elim-avail-ext.	-load-combine	-mod.-debuginfo	-scev-aa
-basicaa	-extract-blocks	-loop-accesses	-nary-reass.	-scoped-noalias
-basiccg	-flattencfg	-loop-deletion	-no-aa	-simplifycfg
-bb-vectorize	-float2int	-loop-distribute	-objc-arc	-sink
-bdce	-functionattrs	-loop-ex.-single	-objc-arc-aa	-slp-vectorizer
-block-freq	-globaldce	-loop-extract	-objc-arc-apelim	-slsr
-bounds-checking	-globalopt	-loop-idiom	-objc-arc-contrac.	-spec.-execution
-branch-prob	-globalsmodref-aa	-loop-instsimpl.	-objc-arc-expand	-sroa
-break-crit-edg.	-gvn	-loop-interchan.	-pa-eval	-str.-d.-proto.
-cfl-aa	-indvars	-loop-reduce	-part.-inl.-libcal.	-str.-dead-d.-info
-codegenprepare	-inline	-loop-reroll	-part.-inliner	-strip
-consthoist	-inline-cost	-loop-rotate	-pl.-ba.-safe.-im.	-strip-d.-declare
-constmerge	-instcombine	-loop-simplify	-place-safep.	-strip-nondebug
-constprop	-instcount	-loop-unroll	-postdomtree	-structurizecfg
-correlated-prop.	-instnamer	-loop-unswitch	-prune-eh	-tailcallelim
-cost-model	-instrprof	-loop-vectorize	-reassociate	-targetlibinfo
-count-aa	-instsimplify	-loops	-reg2mem	-tbaa
-da	-intervals	-lower-expect	-regions	-tti
-dce	-ipconstprop	-loweratomic	-rewr.-sta.-for-gc	

5.3.3 Baseline algorithms

For comparisons with the graph-based approaches, we implemented the sequential insertion algorithm from (Huang et al., 2013) and a Genetic Algorithm (GA)-based approach (see Section 2.5). We used GA with the same configuration as described in (Martins et al., 2014b,a, 2016). We also compared with our SA-based approach (see Section 4.1).

5.3.4 Results for leave-one-out validation

We present and analyze herein the results we achieved from executing the graph-based exploration algorithm with different numbers of iterations; and compare those results with the ones achieved by using the baseline algorithms mentioned in the previous section. Here we present the results achieved using the leave-one-out cross-validation method with the Texas Instruments functions (see Table 5.1). We also characterize the graph structures used in the context of validating our new approach using the leave-one-out method.

We used the performance (i.e., measured in number of CPU cycles) resulting from optimization with the best LLVM Optimizer –OX flag individually found for each function as the baseline for calculating the individual functions speedups when using the specialized sequences found with DSE. We rely on the same graph structure to guide the implementation of the SA extended with compiler pass positioning information (see Section 4.2.7).

Individual performance speedups. Figures 5.11, 5.12 and 5.13 present individual performance speedups obtained with 1,000 and 100,000 compile/simulate evaluation points (phase orders). Compared with other functions, *DSP_mat_trans* has the highest performance improvement. The lowest speedups are obtained for the *IMG_wave_vert*, for all algorithms except SA+Graph (the lowest speedup was reported with *IMG_pix_expand*). For 100,000 iterations, the maximum individual function speedups are 1.85×, 2.22×, 2.13×, 2.24×, 2.02×, and the minimum are 1.02×, 1.02×, 1.03×, 1.06×, 1.05×; for the sequential insertion, GA, SA, SA+Graph and Graph algorithms, respectively. As expected, as the number of iterations decreases, the maximum and minimum speedups decrease. For 1,000 iterations the maximum speedups are 1.59×, 1.90×, 1.87×, 1.90×, 1.92×; and the minimum speedups are 0.55×, 0.82×, 0.78×, 0.97×, 1.01×; for the sequential insertion, GA, SA, SA+Graph and Graph algorithms, respectively. For 100 iterations the maximum speedups are 1.76×, 1.65×, 1.81×; and the minimum speedups are 0.40×, 0.55×, 0.96×; for the SA, SA+Graph and Graph algorithms, respectively.

Geometric Mean Speedups. Figure 5.14 presents the geometric mean speedups over the best per function chosen –OX LLVM optimization level for each DSE algorithm.

Sequential Insertion resulted always in the worst geometric mean performance speedup. GA and SA tend to result in compiled code with similar performance. The former results in better compiled code than the latter (1.14× vs. 1.12×) when the number of iterations is set to 1,000; and the inverse is true for 10,000 iterations (1.20× vs 1.21×) and 100,000 iterations (1.22× vs

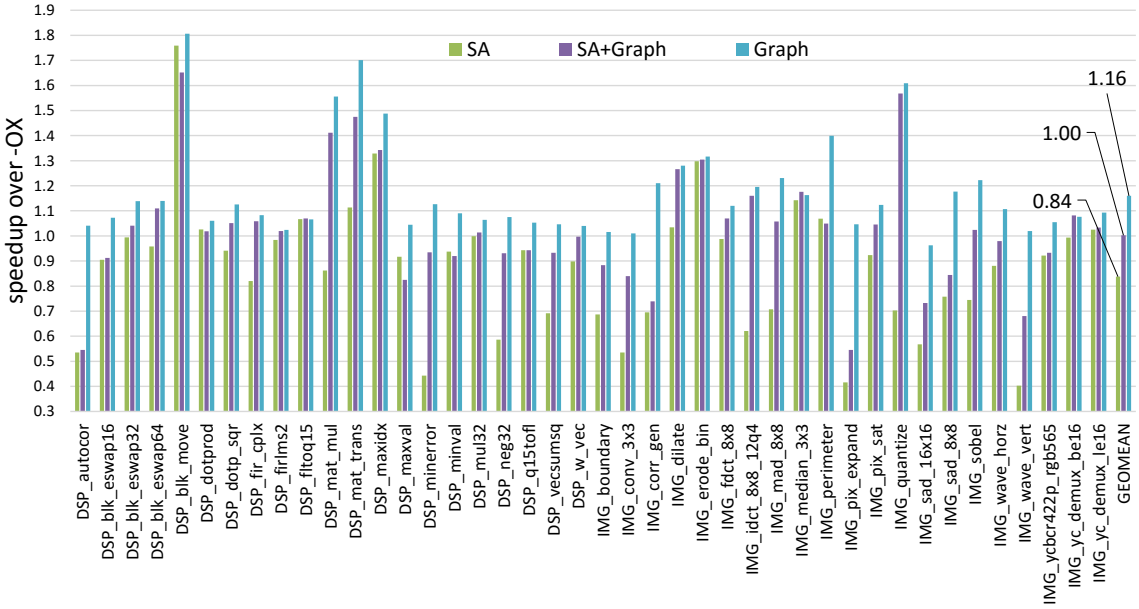

Figure 5.11: Individual speedups over the best per function chosen –OX LLVM optimization level for each DSE algorithm, when exploring 100 phase orders.

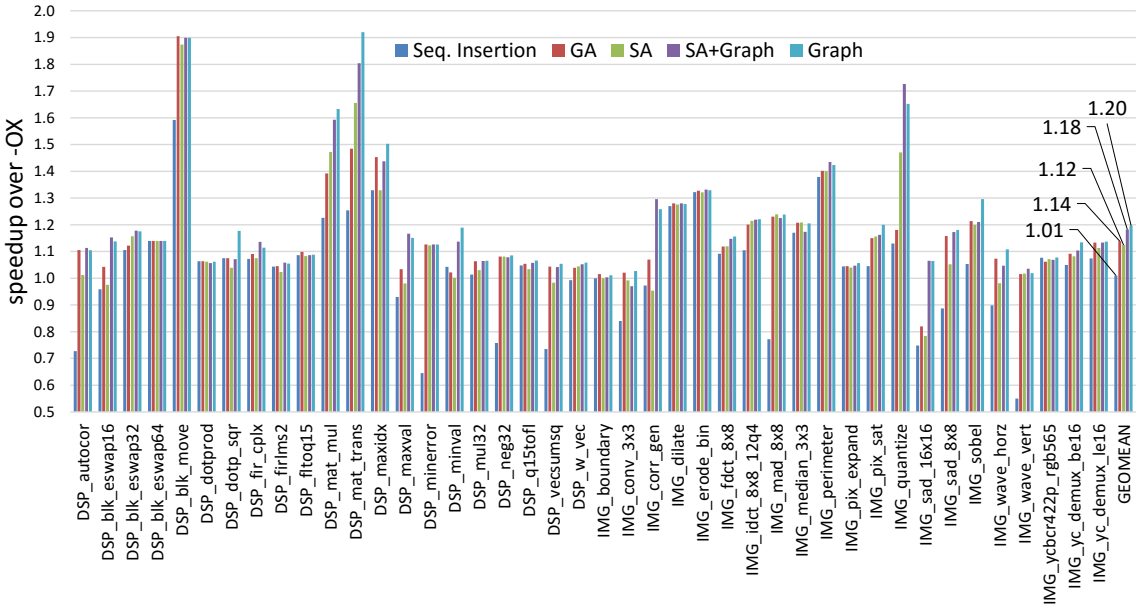

Figure 5.12: Individual speedups over the best per function chosen –OX LLVM optimization level for each DSE algorithm, when exploring 1,000 phase orders.

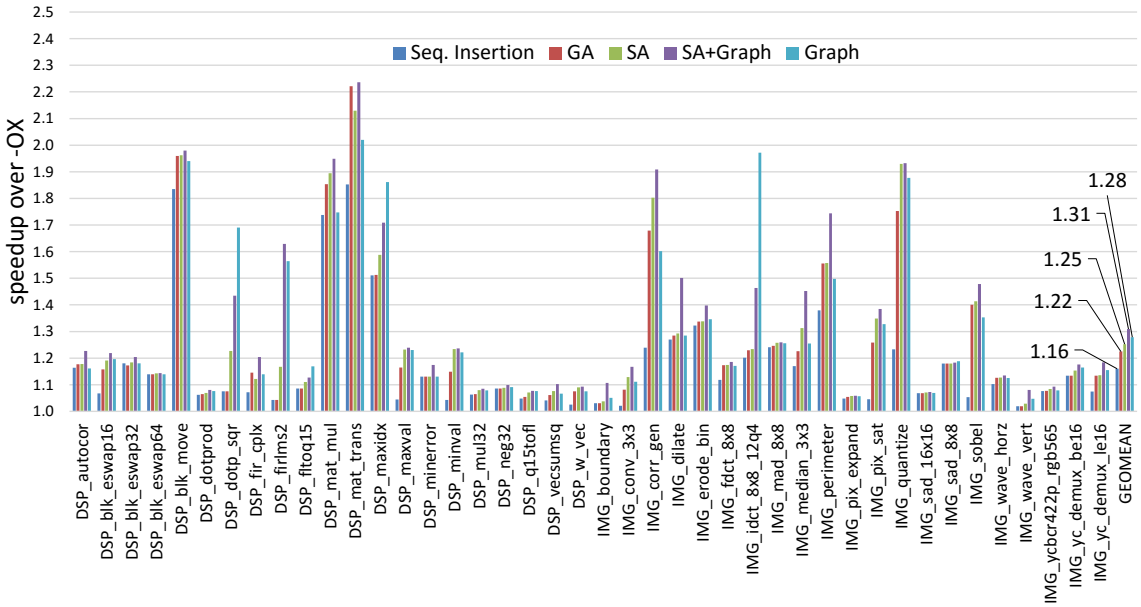

Figure 5.13: Individual speedups over the best per function chosen –OX LLVM optimization level for each DSE algorithm, when exploring 100,000 phase orders.

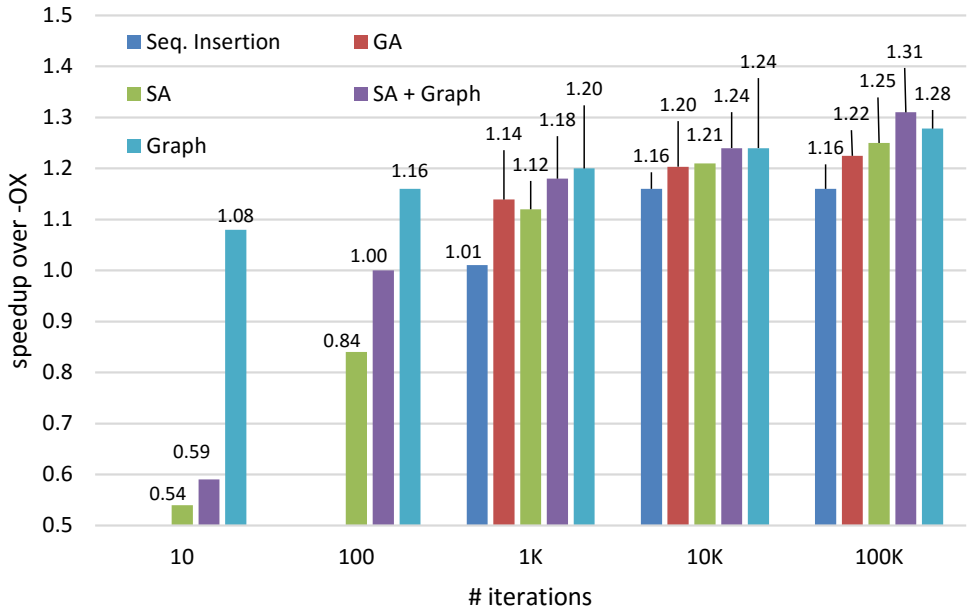

Figure 5.14: Geometric mean speedups over the best per function chosen –OX LLVM optimization level for each DSE algorithm.

1.25×). The SA+Graph always results in a higher geometric mean speedup than the other SA-based method. The new Graph-based exploration algorithm results in the highest geometric mean speedup, for all numbers of iterations considered up to 10,000 iterations.

For 100,000 compile/simulate evaluation points, the geometric mean speedup is 1.16× for Sequential Insertion, 1.22× for GA, 1.25× for SA, 1.31× for the SA+Graph, and 1.28× for the Graph-based approach. For 100 iterations, the new approach resulted in a speedup of 1.16×, while SA and SA+Graph resulted in speedups of 0.84× and 1.00×. There was only a >1 geometric mean speedup over the best –OX with 10 iterations when relying on the Graph-based algorithm, with a resulting speedup of 1.08×.

Given a number of compile/simulate cycles, the GA-based and the non-extended SA-based approach resulted in sequences that when passed to the LLVM optimizer tool result in compiled code with very similar performance. This may indicate that both approaches are well tuned, and that further increases of efficiency when using these algorithms can only be achieved by adding clever heuristics; such as is the case of using the graph structure representing compiler pass transitions to guide the Graph approach.

Best and worst speedups Figure 5.15 depicts the geometric mean speedups calculated using the top 10/20 and worst 10/20 individual function speedups. Considering only the 10 or 20 best individual function speedups for each algorithm, the geometric mean speedups increased substantially. Using the new graph-based approach, when considering the best 10/20 individual speedups the geometric mean speedups increase to 1.31×/1.20× (best 10 / best 20), 1.45×/1.29×, 1.50×/1.34×, 1.62×/1.42× and 1.77×/1.49× (for 10, 100, 1,000, 10,000 and 100,000 iterations), respectively, over the original speedups of 1.08×, 1.16×, 1.20×, 1.24×, and 1.28×.

The speedups achieved by Graph for the 10 and 20 highly improved functions for a very small number of iterations (e.g., 10 and 100) show the importance of the approach and its capability to provide significant performance improvements for a number of benchmarks.

5.3.5 Results for validation with reference functions

Figure 5.16 presents the geometric mean speedups achieved for the Texas Instruments functions presented in Table 5.1 by the Graph algorithm over the best per function chosen –OX LLVM optimization level, when the graph is built using 11 sequences previously found for the set of 11 additional functions. We also include the results obtained using the SA+Graph algorithm using the same graph structure.

When relying on the new Graph-based method for 10 or 10,000 iterations, the compiled functions are not as fast as the programs generated in the leave-one-out experiments (1.03× vs 1.08× and 1.22× vs 1.24×). For 100, 1,000 iterations and 100,000 iterations, the quality (i.e., performance) of the resulting optimized functions was the same; 1.16×, 1.20× and 1.28× respectively.

In the case of the SA+Graph method, speedups increase for 10 (0.81× vs 0.59×), 100 (1.07× vs 1.00×), 1000 (1.19× vs 1.18×) and 10,000 (1.25× vs 1.24×) iterations; and remain unchanged for 100,000 iterations.

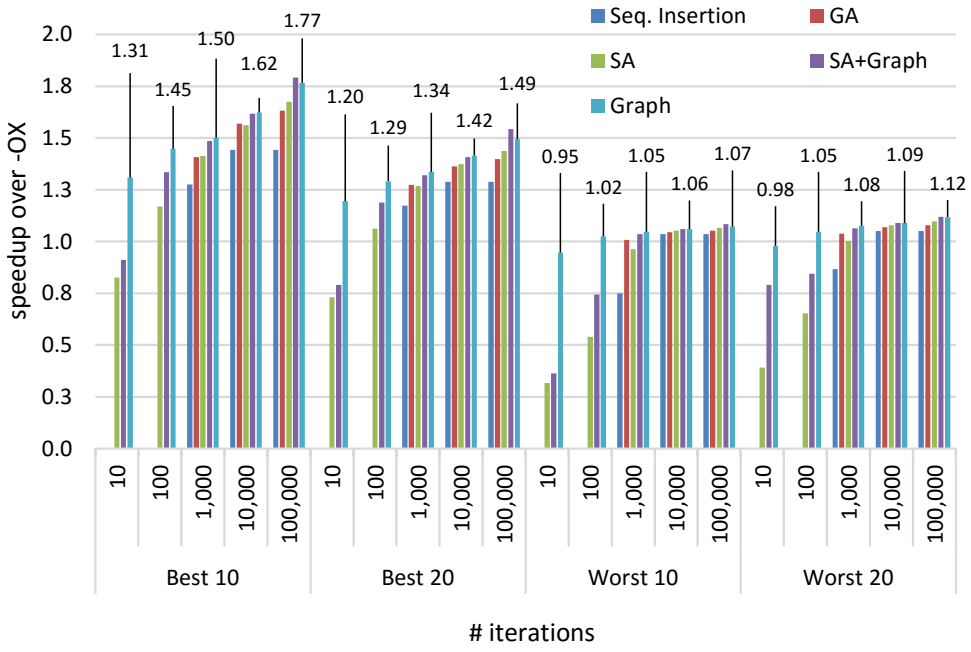

Figure 5.15: Geometric mean speedups considering the best/worst 10 and 20 individual function speedups for different numbers of iterations and different algorithms.

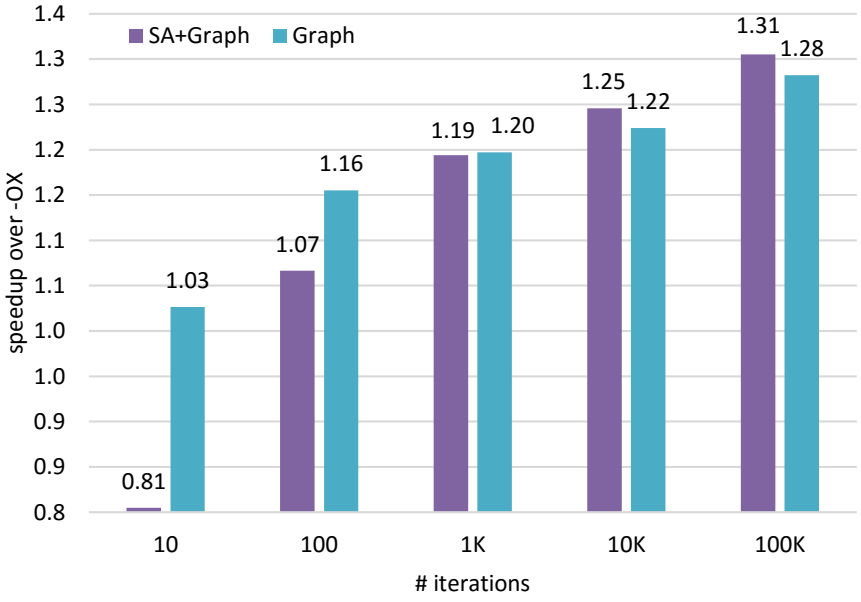

Figure 5.16: Geometric mean speedups over the best per function chosen –OX LLVM optimization level.

The speedups obtained with the SA+Graph and the Graph algorithms are in line with what is achieved in the leave-one-out validation experiments. The graph approach is better than the SA+Graph for 1,000 iterations or less, while the SA+Graph is equal or better for 10,000 or more iterations.

5.4 Targeting a low number of suggestions

This section introduces the results we obtained for our experiments with an approach inspired in the work of Purini and Jain (2013) (see Section 4.3). The results are presented from the point of interest of a prospective compiler user that is interested in optimization trough compiler sequence specialization if it can be achieved with up to 10 additional evaluations (i.e., compilations/measurements). We report the geometric mean improvements achieved with different numbers of additional sequences, from the use of just a single sequence to the maximum number considered for the experiments. In the experiments presented in this section, it is assumed that the optimization level that most aggressively targets the objective metric of interest (e.g., –O3 for performance, –Os for code size) is always evaluated. In case, for a given function/program, none of the compiler sequences additionally evaluated result in improving the function over that optimization level, then the generated solution (e.g., compiled function/program) by the latter (i.e., the –Ox optimization level) is considered over the solutions generated by compilation with the formers (i.e., the additional compiler sequences).

5.4.1 Reference and test functions

We used 30 PolyBench/C kernels as training/reference set and 42 Texas Instruments kernels as test set (see Section 5.1.1). The K sequences are selected based on their effectiveness when compiling the PolyBench/C kernels and are evaluated on the Texas Instruments kernels (our test set).

5.4.2 Target architecture and metric

The experiments presented in this section were performed in the context of compilation for a LEON3 core, using the simulator and configuration described in Section 5.1. Binary execution performance is the target metric that is considered when selecting the K sequences. The procedures presented would work for other metrics (e.g., energy consumption, code size) and/or targets, as long as it is possible to measure the metric of interest for the target of choice in a way that can be integrated in a DSE loop for the generation of the initial data from which sequences for the K set are selected.

5.4.3 Generating table with improvements over baseline

We started by compiling each of the 42 Texas Instruments kernels and each of the 30 PolyBench/C kernels (see Section 5.1.1) with the –O3 flag, which corresponds to the optimization level that most aggressively optimizes for performance. The resulting binaries were executed on a cycle accurate

LEON3 simulator (see Section 5.1.2) and the performance of each binary was measured as the number of CPU cycles needed for the execution of the functions being optimized. LLVM 3.9 was used for the experiments presented in this section, considering the same approach to select what compiler passes to consider for exploration that was used for the experiments presented in the previous section.

We randomly generated 100,000 LLVM compiler sequences, and used each sequence to compile each of the functions. Each resulting binary (after linking with the corresponding `main()` function) was executed. For each execution, a speedup was calculated having as baseline the performance of the binary generated from compilation with the –O3 flag.

The result of compiling all the PolyBench/C and Texas Instruments kernels using those sequences is a table containing $100,000 \times (30+42)$ real numbers, each representing a speedup; where each column represents a function and each row represents a compiler sequence. The table was then separated, so that the values representing the speedups over –O3 from compilation of the 30 PolyBench/C kernels with each of the 100,000 sequences were used as data to select the K sequences (i.e., served as reference/training data), and the data for the Texas Instruments kernels was used to validate the approach (i.e., check what speedups were achieved based on what sequences were selected to be part of the K set).

5.4.4 Selecting and using the sequences

We set K to 20 for the experiments presented here, in order to evaluate what happens when more sequences than our target for exploration (i.e., 10 or less) are considered.

We experimented with a parameter (see Section 4.3.3) that is not part of the approach presented by (Purini and Jain, 2013). This new parameter represents the maximum distance the geometric mean speedup (in the reference/train set of functions) achieved by compiling with a given sequence is from the geometric mean resulting from compiling with the sequence that results in highest geometric mean (represented in the speedups table), that still allows it to be selected for the set of K sequences. Sequences that are not within a maximum distance factor are automatically disqualified from being considered for the K set. For instance, if $\delta = 0.05\times$, the only sequences considered are the ones resulting in $0.95\times$ (i.e., 95%) or more of the geometric mean speedup achieved with the sequence resulting in highest geometric mean speedup. We present results for when δ equals $0.03\times$, $0.05\times$, $0.1\times$, $0.15\times$, $0.2\times$, $0.25\times$, $0.3\times$, and $0.5\times$.

Figures 5.17 and 5.18 present results for the use of different values for δ, for a number of compilations/evaluations up to K. Sequences from the K set are used for compilation/evaluation in order (i.e., from sequence 1 to K). The way the extraction of the K sequences is performed results in sequences that have a lower index (i.e., sequences extracted first) providing a higher coverage of the function space of the reference functions (see Section 4.3.2), and consequentially are also more likely to be suitable to compile new functions. For instance, sequence with index x is more likely to be more suitable to compile both functions from the reference set and new functions than sequence with index $x+y$, where y is any positive integer up to $K-x$.

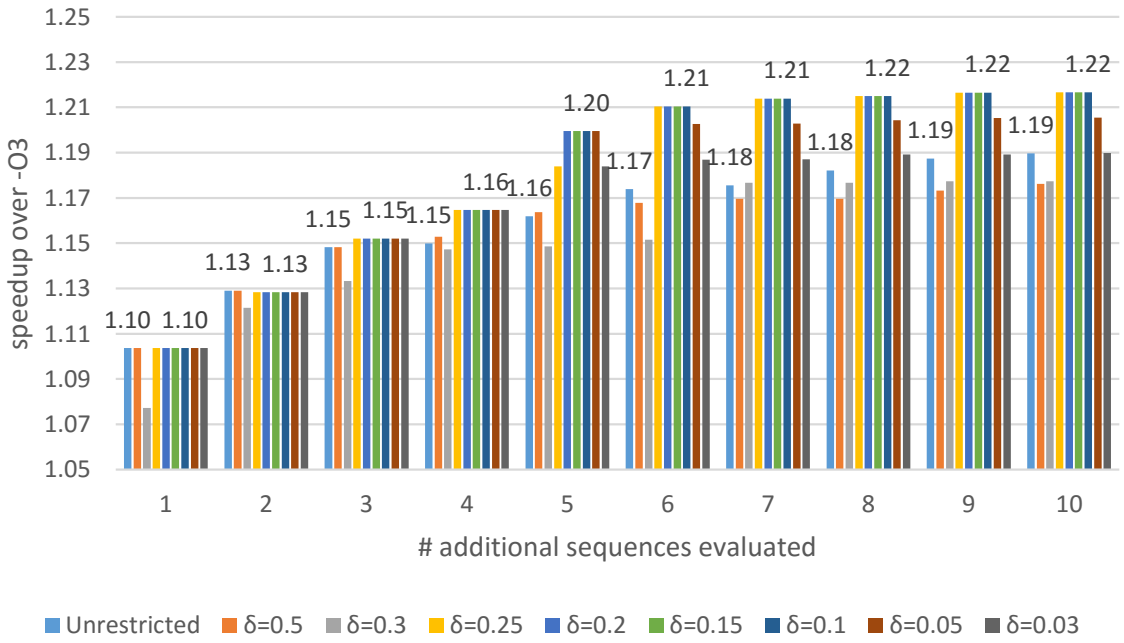

Figure 5.17: Speedups considering optimization of all the functions from Texas Instruments and the use of a parameter that avoids overfitting the K sequences to the PolyBench/C reference functions (evaluating up to 10 sequences from the K set).

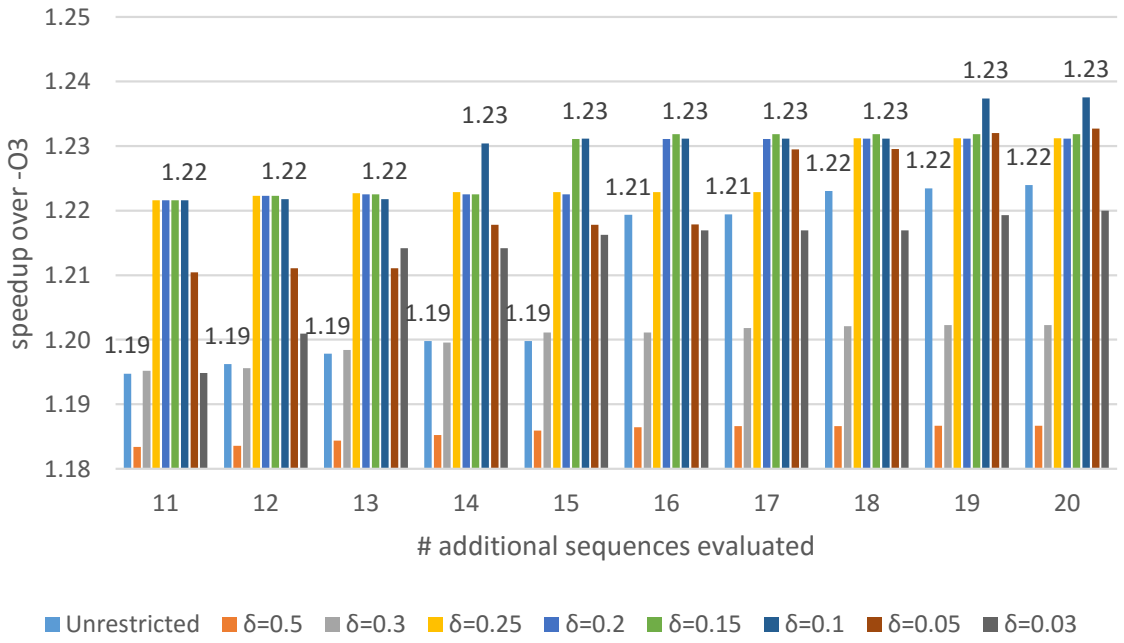

Figure 5.18: Speedups considering optimization of all the functions from Texas Instruments considering the the use of a parameter that avoids overfitting the K sequences to the PolyBench/C reference functions (evaluating from 11 to 20 sequences from the K set).

The results show that the sequences selected for the K set based on their performance on the PolyBench/C kernels (i.e., the reference/training set) are suitable to compile the Texas Instruments kernels. Geometric mean performance improvements were always achieved in comparison with the use of the –O3 flag, even when evaluating only a single compiler sequence from the K set. This result is quite relevant, in the sense that the reference and the test sets are composed of functions from different benchmarks, with a number of differences. For instance, the 30 PolyBench/C kernels are very floating intensive while the 42 Texas Instruments kernels are dominated by integer computations.

The geometric mean improvement over –O3 on the 42 Texas Instruments kernels (i.e., test set) increased with the use of the δ parameter to restrict the sequences that can be selected to be part of the K set. Compared with the unrestricted selection of K sequences, the use of the δ parameter only resulted in deteriorating execution performance on the Texas Instruments Kernels (test kernels) for $\delta = 0.3\times$ and for $\delta = 0.5\times$. Setting δ between $0.05\times$ and $0.25\times$ consistently resulted in binary execution performance improvements over not using the δ parameter to condition selection for the K set (i.e., unrestricted selection, which is equivalent to $\delta = +\infty$) when using the resulting K sequences to compile the Texas Instruments kernels. This is caused by reducing the propensity to overfitting the PolyBench/C functions when selecting the K sequences from the initial set of $100,000$ sequences. For instance, although without applying any restriction with the δ parameter to the selection of K sequences the performance of the sequences selected for the K set is higher ($1.36\times$ over –O3) in the reference functions than if $\delta = 0.1$ ($1.33\times$ over –O3), the performance on the set of functions used for testing is higher in the later case.

Figures 5.19 and 5.20 compare the approach based on K sequences (for $\delta = +\infty$ and for $\delta = 0.1$) with the IterGraph approach constructed using different sets of sequences, and with random selection and code feature-based selection (using MILEPOST features (Fursin et al., 2008)) of the K sequences extracted with $\delta = 0.1$.

When considering a very low number of phase orders to be suggested, the IterGraph approach tends to not perform as well as the approach based on the use of the K sequences. For achieving a geomean speedup of $1.21\times$ over –O3 the IterGraph approach required at least 16 additional evaluations (i.e., on top of evaluating –O3) for the graph configuration that experimentally achieved this performance in the least evaluations, which used the graph built from the $K = 20$ sequences that were extracted using the approach inspired in Purini and Jain with no restriction. The approach inspired in Purini and Jain, when relying on K sequences extracted with $\delta = 0.1$ only required evaluating 6 sequences from the K set in order to achieve a speedup of $1.21\times$. These results show that this approach, with the addition of the δ parameter, can be specially interesting when considering a very low number of phase orders to evaluate. For the experiments considered here, it was able to consistently achieve the highest geomean speedups when evaluating 3 or more compiler phase orders from the K set. For 1 and 2 evaluations, the IterGraph approach resulted in higher geomean binary execution performance, using the 30 sequences found to be better suited for the 30 PolyBench/C kernels (i.e., one sequence per kernel) and using the 20 sequences extracted without restriction to generate the graph representing favorable phase orders, respectively. The other

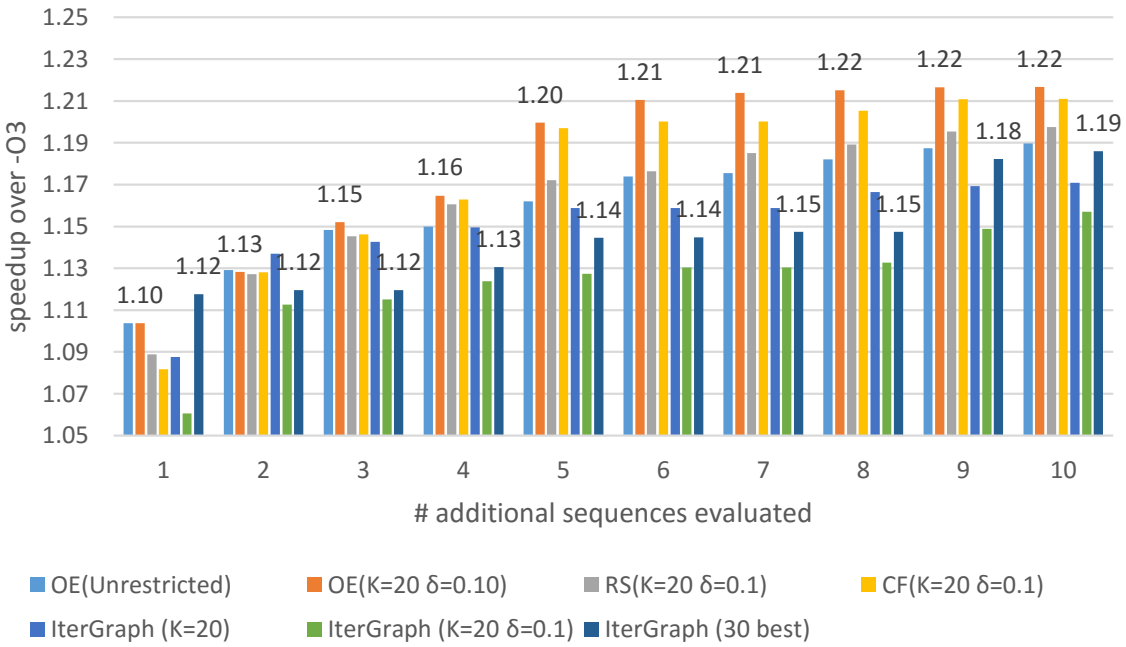

Figure 5.19: Comparing with the IterGraph approach (evaluating up to 10 sequences from the *K* set). OE: Ordered Evaluation. RS: Random Sampling. CF: Code Features.

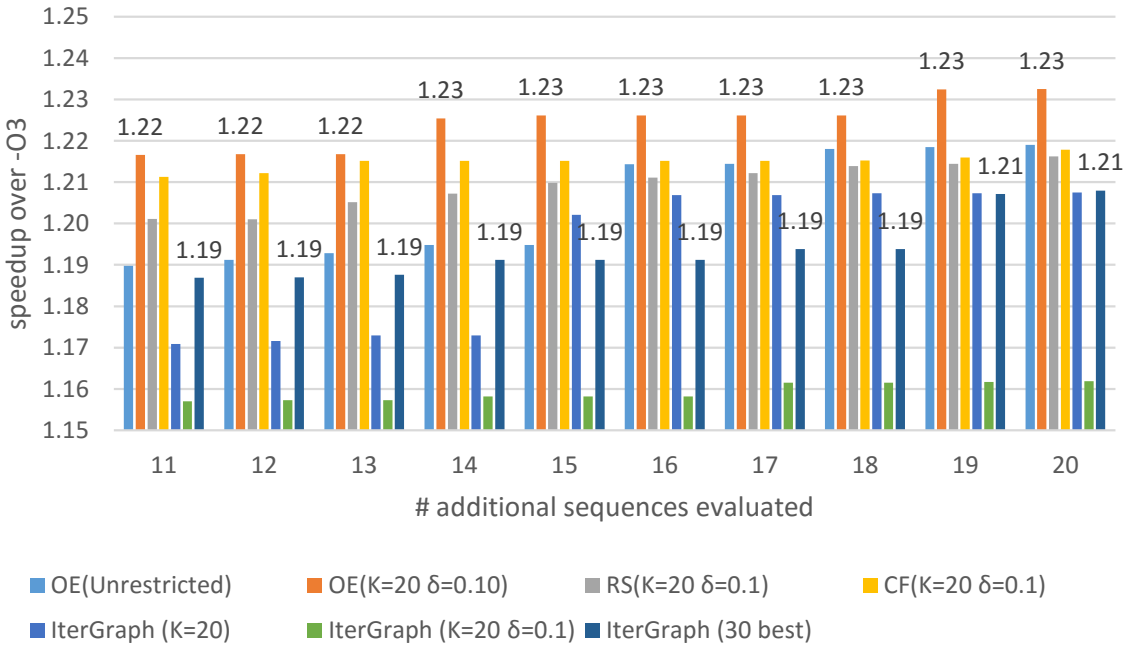

Figure 5.20: Comparing with the IterGraph approach (evaluating from 11 to 20 sequences from the *K* set). OE: Ordered Evaluation. RS: Random Sampling. CF: Code Features.

IterGraph configuration represented in Figures 5.19 and 5.20, i.e., using 20 sequences extracted with $\delta = 0.1$ resulted consistently in generating sequences that result in worse binary performance than any other approaches.

Exploration using the IterGraph approach continues to be worthwhile in the case of exploring more iterations. For 100 exploration iterations the IterGraph approach achieves 1.24×, 1.25× and 1.25×, if the graph representing the favorable compiler subsequences is constructed with the 30 best sequences in the set of 100,000 (one per PolyBench/C kernel), with the $K = 20$ sequences extracted using $\delta = +\infty$, and with the $K = 20$ sequences extracted using $\delta = 0.1$, respectively. For 1,000 iterations the speedups improve to 1.28×, 1.29×, 1.27×, respectively.

5.5 Summary

We presented results targeting a number of different microarchitectures in the context of phase selection and ordering exploration with LLVM as a means to optimize functions. Geometric mean performance improvements of up to 1.2× were achieved when comparing with the execution performance achieved when relying only on the compiler optimization level flags (related with research question *Q1*). The results suggest that the different targets have different optimization potentials (related with research question *Q2*). Additionally, we presented results of using sequences on a given target that were found previously for other targets. Using sequences on a target that were found for other target resulted in a considerable loss of execution performance (related with research question *Q3*). For instance, using sequences found for the 2-stage Mor1kx in the ARM Cortex-M4 results in a reduction of 25% in relation to the geometric mean speedup achieved when exploring compiler optimization sequences directly for the ARM Cortex-M4. As expected, this only seems to work well if the two architectures are very similar (3-stage MicroBlaze and 5-stage MicroBlaze).

We presented results that compare the IterGraph approach to other iterative approaches for DSE, including a sequential insertion based algorithm, a GA and two SA-based algorithms (related with research questions *Q5*); one of which relies on the same graph structure used by IterGraph. The results strongly show that the graph-based approaches significantly outperform the other approaches evaluated, as they are able to achieve the phase orders resulting in similar performance but with a significant reduction of the number of DSE iterations (related with research question *Q7*); especially when considering 100 or less exploration points. When targeting a set of 42 image and digital signal processing functions to a LEON3, the IterGraph algorithm is able to consistently find a compiler phase order better than the best LLVM standard optimization levels in only 10 compile/simulate iterations; while none of the evaluated algorithms were able to achieve that. Executing for only 100 iterations this algorithm suggested compiler phase orders achieving a geometric mean speedup of 1.16× (both cross-validated and non-cross-validated experiments) over the best individually (i.e., per function) found –Ox flag, while the closest method tested was the SA+Graph with 1.07× speedup (for the non-cross-validated experiments).

We presented our experiments with an approach that results in the generation of a considerably small set of K sequences that are to be evaluated by the compiler user (manually or in an automated way, if the compiler includes a DSE loop) in addition to the compiler flags that typically better optimize the metric of interest (e.g., –O3 for performance, –Os for code size). We experimented with approaches for extracting a set of K sequences, from a set of $100,000$ sequences previously generated and evaluated, based on their performance covering a program space (i.e., a set of reference functions), and evaluated their effectiveness for compilation of other functions. We present the results of experiments to assess the effect of a parameter that helps improving the effectiveness of the K sequences when used to compile a new program/function, by only allowing sequences that result in a geometric mean speedup (in the reference set) at least of a factor (given by this parameter) of distance from the highest geometric mean speedup to be eligible for the K set. This approach was evaluated and compared with the IterGraph approach with the purpose of accessing the improvements that can be obtained when only relying on the evaluation of a considerably small set of sequences (related with research question *Q8*). Tuning this parameter resulted in a geometric mean speedup of the same 42 image and digital processing functions by a factor of $1.2\times$ over –O3 while only relying on testing only 5 compiler sequences, on top of evaluating –O3. Not using this parameter when selecting the K sequences resulted in 16 compilations/executions being required (in addition to –O3). Only 8 additional compilations on top of –O3 were required to achieve a speedup of $1.22\times$, for a arguably wide range of values for this parameter. Not using this parameter required 18 additional compilations/executions to achieve the same geometric mean speedup. The IterGraph approach required a minimum of 15 additional compilations to achieve a geometric mean performance improvement of $1.20\times$.

Chapter 6

Conclusion

In this chapter we present our conclusions about the work presented in this thesis, as well as future work.

6.1 Final remarks

We developed a Design Space Exploration (DSE) system to satisfy our need of evaluating different approaches for exploring compiler sequences, i.e., selection of compiler passes and the order of their application in optimizing compilers (see Chapter 3). In addition, our DSE system allows the reuse of the same DSE algorithm specifications along with different optimization purposes, with different compilers and different target architectures such as Central Processing Units (CPUs), Graphics Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAs) (if supported by the compiler backend used). The current version supports reusing the specifications of metrics for optimization strategies (e.g., energy/power minimization, performance maximization), target interfaces (e.g., specification about how tools, such as simulators or hardware platforms, for a given target microprocessor, are executed). The different components connect with the DSE system core through a plugin approach, not requiring additional code for making new DSE schemes, metrics, or targets, available to the DSE system through the user interface.

We demonstrated experimentally, across a considerable number of different microprocessor/microcontroller targets, that given a program kernel and a target platform, better performance than when using the default settings or any of the available optimization levels can be achieved by tuning a compiler sequence specifically for each program kernel and microprocessor/microcontroller target pair. Using our DSE system, we could find compiler sequences with higher performance than the compiler sequences that represent the traditional optimization strategies (e.g., the –O3 flag). Our system found better sequences for most program codes, when targeting the LEON3 processor, the main target of our phase ordering exploration experiments, and when compiling for other microprocessor/microcontroller targets, including OpenRISC Mor1kx (Baxter and Kristiansson), MicroBlaze, ARM and LEON3 CPUs. Our Simulated Annealing (SA)-based DSE scheme was able to generate sequences that result in considerably higher binary execution performance

than the traditional –Ox flags even when considering only compiler passes present in the compiler sequences represented by the same –Ox flags. Using LLVM, we were able to achieve a geometric mean speedup between 1.08× (ARM Cortex-M4) and 1.20× (OpenRISC Mor1kx 6-stage) versus the performance achieved with the best experimentally tested –Ox flag for each program, when targeting a set of Texas Instruments kernels for embedded systems to different microprocessors. In addition, the results of these experiments also show that using the best sequences found for one of the microprocessors/microcontrollers in other microprocessor/microcontroller results in the generation of less optimal code.

A number of other metrics can be improved by selecting which passes are executed and in which order for each individual function/program and target pair. Performance is only one of such metrics that can be improved by using our methods and tools. Smaller code size, and less energy and/or power consumption can be achieved by using our exploration system and DSE schemes; given the required source/target-specific tools (e.g., frontends, tools for measuring energy, cycle-accurate simulators) are added to the DSE loop in our compiler sequence exploration system.

Although the overhead of finding compiler sequences alternative to the standard optimization levels (usually represented by –Ox flags) using pure iterative based approaches (e.g., SA-based approach) presented in this thesis can be considered intolerable for some users, there are cases where it would be perfectly tolerable. Waiting minutes, hours, or even days, for achieving an improvement as small as 5% (we found compiler sequences leading to over 2× performance improvement) can be justifiable in some scenarios. For example, if an application is being compiled for a product that is expected to sell millions of units, the product designers can save on hardware costs if the compiled code performs better, uses less memory, or saves energy; by using a slower and cheaper processor, a smaller and cheaper memory chip, or a smaller and cheaper battery. Or alternatively, they can make a better product to their end-users, using the same hardware. Additionally, in some cases, the overhead added by the compiler exploration time would account for very little time when compared with the other tasks of the product development cycle. Compiler phase sequence specialization, both in the form of phase selection and/or ordering, can be specially useful for optimizing the types of programs typical of embedded systems and other domains, such as High Performance Computing (HPC), as they tend to have very localized hotspots. In situations that make DSE more expensive (e.g., kernel takes too long to execute with original input parameters), there are a number of techniques that can be used to make exploring sequences for a given function/program more efficient.

We approached the challenge of compiler phase selection/ordering in a way that strives to reduce the overhead of finding suitable compiler sequences, so that it is acceptable in a larger number of use cases. We presented results generated by approaches that considerably reduce the number of exploration points, while being able to find sequences that have quality comparable with the sequences found using other DSE schemes, such as our SA-based DSE scheme, Genetic Algorithms (GAs) and and other enumeration based approaches.

We developed an approach that relies on generating a graph representing favorable compiler pass transitions. This graph can be generated from sequences of compiler passes previously gener-

ated with other iterative methods for a reference set of functions/programs, or it can be generated by compiler experts. The graph nodes represent individual compiler passes or subsequences, and directed edges connecting the nodes represent the order in which passes/subsequences are typically executed. Different graphs can be created for different domains (e.g., automotive, multimedia) so that better code can be generated with that practice (will require mapping any given new program to one of the existing groups, e.g., automotive). New sequences are generated sampling the graph, or by using the graph structure to focus exploration in other algorithms. The graph has been experimentally used in two approaches: (a) extending the SA-based DSE scheme (SA+Graph), and (b) using the graph to directly generate compilation sequences (IterGraph). The use of the graph structure makes SA+Graph converge faster to solutions of comparable quality. The use of the IterGraph approach and the SA+Graph resulted in finding compiler sequences that result in high geometric mean speedups faster (i.e., requiring less DSE iterations) than the other approaches evaluated, which include other iterative approaches from the state-of-the-art. The IterGraph and the SA+Graph approaches also perform better than the other enumeration-based approaches evaluated for considerably high numbers of DSE iterations (e.g., $100,000$), achieving geometric mean speedups of $1.28\times$ and $1.31\times$ in our leave-one-out experiments over the best individually found standard optimization level, respectively.

We suggested an extension for the IterGraph approach using code features. In this approach, the new sequences being generated for a new function/program reflect its similarity with the functions/programs from a set of reference functions. We propose the use of the cosine distance between static and/or dynamic code feature vectors as similarity metric, including a way to weight the influence of the sequences for the reference functions/programs differently based on their cosine distance to the new function/program. The use of this metric seems to have potential, based on our experiments in the context of compiler sequence suggestion based on static code features in OpenCL kernels (see Annex C).

We worked towards evaluating and improving an approach that might be more suitable to more compiler users, and that also has some key advantages over other more traditional enumeration-based approaches explored in this work. Given a large number of sequences and their impact on a set of reference functions/programs, the approach consists in extracting a small number of sequences (we presented results for up to 20 sequences) that cover the function/program space of the reference functions, while avoiding overspecialization to that function/program space. We adapted a method from Purini and Jain (2013) in a way that, although the impact of those sequences decreases on the reference functions, better performance is achieved when they are used to compile new functions/programs. By requiring this set of sequences to be less specialized for the reference functions, we could significantly improve their impact on new functions/programs. Evaluating only 5 from these sequences, the performance of a set of new functions improved by a factor of $1.2\times$ over –O3, and with 10 additional sequences it improved by $1.22\times$. This suggests that the method can be quite effective at improving new functions/programs while relying on evaluating only a small set of compiler phase orders. If overspecialization to the reference functions is not taken into consideration, then the improvements with 5 and 10 compilations using these sequences

are reduced to 1.16× and 1.19× for the new functions, respectively. This is still far from what is achieved with iterative compilation with generation of sequences (1.29× for IterGraph using the same target platform and LLVM version), but it may still be interesting for a number of compiler users (see Section 4.3.5).

6.2 Future work

Future work regarding the graph-based approaches includes evaluating schemes that prune the design space using code features. The use of program/function features from the kernels for which compiler sequences may have the potential to reduce exploration time while achieving functions/programs of similar quality. For instance, there is no need to execute loop related compiler passes (e.g., loop unrolling, loop invariant) if a kernel has no loops or to execute the constant propagation compiler pass if the code has no constants. A challenge for the use of code features in the context of compiler sequence exploration including phase ordering is related to the fact that the compound effect of the application of a chain of compiler passes is difficult to predict if only the source code of the function/program is taken into account. We believe that considering only features in the source code can limit the potential of graph-based approaches. Instead, taking into account features at the intermediate representation (IR) of the function/program being optimized allows to probe the changes made by each of the compiler passes (over the IR transformed by the passes executed previously). Looking at the transformed IR at each step of the compiler sequence construction (i.e., after adding each pass to the sequence under construction) in order to influence the selection of the next pass may allow the extraction of optimization potential that is difficult to leverage if only static features in the original source code or in the IR generated from the source code without any transformation applied are considered. Still in the context of DSE using the graph, we also plan to perform a comparison between different possible approaches to deal with parameterizable compiler passes, that are able to efficiently work with the increased solution space while avoiding too much specialization to a reference set of functions/programs.

The use of the clustering-based approach described in (Martins et al., 2014b,a, 2016) is orthogonal to the SA+Graph and the IterGraph approaches presented in this thesis, and both can be combined. Clustering can be used as an approach to remove nodes from the graph that represent the use of compiler passes that are not associated with the cluster of functions/programs selected for a given input program/function.

Given the strong presence of GPUs (e.g., in HPC, desktop, mobile) and the increasing relevance of GPUs as a way to achieve higher absolute throughput and higher throughput per energy unit, we have started performing DSE targeting GPUs while having energy consumption minimization as target optimization metric. The goal is to characterize the design space and derive DSE schemes specialty tailored to search for compiler sequences that optimize for energy consumption.

Research work towards deriving a small set of compiler sequences (e.g., 10) targeting a GPU or a family of GPUs is, we believe, an interesting and promising line of research. This could entail

repurposing our experimental setup concerning our experiments that we performed targeting the LEON3 in the context of the Ph.D. work. In order to generate the initial large set of sequences targeting a GPU instead of the LEON3, our DSE system only requires changing a flag representing the exploration target. The remaining steps to extract a set of relevant sequences from the initial large set of sequences is the same, which is already automatized. The real challenge will be to find a set of OpenCL kernels that is representative of most types of OpenCL kernels compiled to GPUs. Based on the performance improvement potential that we experimentally demonstrated that is available by phase selection and ordering when targeting GPUs (see Appendix C), we believe that a list of sequences specially tailored for the most common GPU architectures would generate interest. Given that enough interest is generated, those sequences could even be included in the compilers targeting those GPU architectures as additional optimization levels.

Another path is to research the use of function/program features for suggesting sets of compiler sequences, as an extension to the approaches proposed with focus on generating improved code faster. We envision an approach, where given a set of function/program features of a new function/program, instead of generating new sequences by online phase selection and/or phase ordering, a different set of K (e.g., $K = 10$) sequences would be used based on the the similarity with a set of reference functions/programs. This contrasts with the the approach we presented in this thesis, in which only a single set of K sequences representative of a set of functions/programs is extracted offline to be used by the compiler end user (e.g., in the form of extra –Ox flags). In contrast, in this approach, multiple K sets can be extracted offline, each for different points in a multidimentional grid representing a number of combinations of possible distances to a possible new function/program. It is not feasible to extract the K sequences for each of the points in this grid, even if considering intervals of 0.01 from 0.95 to 1.0 in the distance metric (e.g., cosine similarity of feature vectors), because of the fact that 6 distance values per function/program in the reference set would mean it would result in having to extract 6^n K sets of sequences, where n is the number of reference functions/programs. As a workaround, extracting the K sets of sequences associated with a sufficiently large number of points from this space may provide a suitable approximation. For each of the points considered in this grid of distances, the K sequences can be extracted with methods used in the experiments we presented. The difference would be that different weights would be given to the individual functions/programs, therefore a weighted geometric mean would be used when extracting the K sequences covering the reference function space. This results in giving more importance to some codes over others, by a factor determined by the similarity distance points in the grid (and a scaling factor). The user of such code-feature aware compilation toolchain would only need to compile and evaluate with the sequences of the K set suggested based on the code features of the new/function program being compiled. The compiler would select the set of K associated with the point in the multidimentional grid of distances that is closer to the one representing the distances calculated between the new program/function and the program/functions in the reference set. These sequences could be presented to the compiler user as new –Ox levels that would have the particularity of not only being specialty tailored to the target architecture, but also to the code features of the program/function being compiled.

A research path with possible high impact is the phase ordering specialization for compilation of OpenCL kernels generated by higher-level languages. For example, in the context of the MATISSE (Bispo et al., 2013) MATLAB to C and OpenCL compiler. The MATISSE OpenCL code generator supports the generation of OpenCL kernels specialized for particular CPUs, GPUs and FPGAs. Additionally, through specialization, a large number of considerably different C and OpenCL versions can be generated from the same MATLAB functions source code, such as versions with loop interchange turned off or different parallelization distribution mechanisms (i.e., the equivalent of OpenMP *schedules*). This allows for the creation of a dataset of C and OpenCL kernels (both supported by our DSE system) large enough to allow the generation of better models correlating features with the specific phase orders or specific code-dependent design space pruning approaches.

Appendix A

Compiler Phase Ordering as an Orthogonal Approach for Reducing Energy Consumption

Compiler writers typically focus primarily on the performance of the generated program binaries when selecting the passes and the order in which they are applied in the standard optimization levels, such as GCC –O3. In some domains, such as embedded systems and High-Performance Computing (HPC), it might be sometimes acceptable to slowdown computations if the energy consumed can be significantly decreased. Embedded systems often rely on a battery and besides energy also have power dissipation limitations, while HPC centers have a growing concern with electricity and cooling costs. Relying on power policies to apply frequency/voltage scaling and/or change the CPU to idle states (e.g., alternate between power levels in bursts) as the main method to reduce energy leaves potential for improvement using other orthogonal approaches. In this work we evaluate the impact of compiler pass sequences specialization (also known as compiler phase ordering) as a means to reduce the energy consumed by a set of programs/functions when comparing with the use of the standard compiler phase orders provided by, e.g., –OX flags. We use our phase selection and ordering framework to explore the design space in the context of a Clang+LLVM compiler targeting a multicore ARM processor in an ODROID board and a dual x86 desktop representative of a node in a Supercomputing center. Our experiments with a set of representative kernels show that there we can reduce energy consumption by up to 24% and that some of these improvements can only be partially explained by improvements to execution time. The experiments show cases where applications that run faster consume more energy. Additionally, we make an effort to characterize the compiler sequence exploration space in terms of their impact on performance and energy.

A.1 Mapping programs with energy concerns

Mapping applications efficiently is very important when targeting systems with strict requirements (e.g., embedded systems, HPC) , such as energy/power, performance, memory and/or storage. Software optimization driven by an optimizing compiler helps to comply with requirements while using less resources in the process, contributing to the reduction of hardware costs and/or improving user experience.

We experimentally show that optimizing for performance through compiler-driven software optimization as a means of optimizing for energy does not always lead to the most energy efficient compiled functions/programs. We achieve this goal by compiling multiprocessor-ready versions of 12 PolyBench/C 4.1 (Pouchet et al.) kernels, to a dual Intel Xeon workstation, representative of a supercomputer node and to an ARM-based ODROID XU+E board, representative of hardware present on a mobile phone or tablet, and comparing how execution time and energy consumption are affected for the execution of the binaries generated from compilation with phase orders generated by a design space exploration (DSE) method. The results show that although improving performance tends to improve energy efficiency, there are a number of situations where the compiler sequences that result in the best performance do not translate into the best energy consumption. In such cases, there are compiler sequences that allow achieving even better energy savings.

The next sections explain the methodology of the experiments, we present the experimental results and we make an effort to explain them.

A.2 Experiments

We performed a number of experiments in two relevant target platforms in order to evaluate the impact of compiler optimizations using specialized compiler phase orders on both energy and performance.

A.2.1 Platforms

We consider two systems. A workstation with two Intel Xeon E5-2630V3 CPUs (@2.4 - 3.2 GHz Turbo), and 128 GB of DDR4 (@2133 MHz), representative of a supercomputer node; and an ODROID XU+E single board computer (Hardkernel), with a Samsung Exynos 5410 SoC (part of the Exynos 5 Octa series), the same SoC in the the Samsung Galaxy S4 smartphone. The Xeon E5-2630V3 is an X86-64 CPU with Intel's latest microarchitecture for the workstation/server market, the Haswell-EP microarchitecture. The Exynos 5410 SoC on the ODROID includes a Cortex-A15 1.6 GHz quad core CPU and a 1.2 GHz Cortex-A7 quad core CPU, in a configuration referred to as *big.LITTLE*, and 2 GB of LPDDR3 on the same package. The *big* cores are designed for maximum compute performance, while the *LITTLE* cores are designed for maximum power efficiency. Unlike a traditional 8-core CPU (or a dual 4 core), the *big.LITTLE* configuration means that the *big* (Cortex-A15) and the *LITTLE* (Cortex-A7) CPU cores take turns to execute a task,

which is migrated between the two types of cores during execution, in a joint effort to make computation more power and energy efficient.

The operating system for both platforms is Ubuntu. On the dual Xeon-based workstation we use a 64-bit Ubuntu 16.04 LTS system with Linux kernel 4.4.0, and on the ODROID board we use Ubuntu 14.04.2 LTS with Linux kernel 3.4.75.

For experiments with OpenMP, we use version 3.7.1 of the LLVM OpenMP runtime (LLVM Developer Group, d) in both platforms.

For the dual Xeon E5 V3 platform we consider the default power settings with Turbo-boost activated to effectively drive some of the CPU cores up to 3.2 GHz from the base clock of 2.4 GHz. For the ODROID XU+E board we use the two types of ARM cores (Cortex-A15 @1.2GHz and Cortex-A7 @1.6 GHz) in the *big.LITTLE* configuration.

In both cases, frequency voltage scaling is activated and managed by the default power governor on the Linux distributions used; the *powersave* power governor for the Dual Xeon and the *on-demand* power governor for the ODROID.

A.2.2 Functions

In these experiments we use 12 kernels from PolyBench (Pouchet et al.) (version 4.1), and generated parallel versions of the kernels with PLUTO (Bondhugula et al., 2008), a tool for automatic parallelization. PLUTO relies on a polyhedral model generated from the input function/program, specially the parts concerned with loops and operations with arrays, as an abstraction to safely perform parallelization of loops thorough annotating the code with OpenMP pragmas (coarse-grained parallelism) and apply other high-level transformations, such as tiling loops to improve locality (reduces cache misses), and vectorization (i.e., use of SIMD units such as AVX). The parallel versions were generated with PLUTO from versions of the PolyBench/C benchmarks that use dynamically allocated memory. The input dataset was changed for each benchmark to a customized dataset so that the execution time is larger than a threshold required for correct measurement of energy consumption using one of the targets we experimented with (an ODROID-XU+E board).

Table A.1 depicts the functions (and input data) used for the experiments and their number of lines of code (excluding comments) for both the original C version and the OpenMP annotated versions generated using PLUTO. Parallel version of functions identified by an asterisk (*) were generated with loop tiling (*–tile* option) in addition to parallelization (using the *–parallel* option), which annotates the code with OpenMP pragmas.

A.2.3 Energy and performance measurements

We measure the energy consumed by both CPU and RAM on each platform. The energy values reported are always for the sum of both.

For accuracy and higher statistical significance the metrics reported are the result of averaging 30 executions of each compiled function/program. Multiple executions are required because the

Table A.1: Description of 12 PolyBench/C 4.1 functions, input parameters, and lines of code for the original implementations and OpenMP versions generated with the PLUTO automatic parallelizer.

Function	**Description**	**Input**	**CLOC**
2mm	Autocorrelation of an input vector.	*ni: 400, nj:450, nk:550, nl: 600*	11 / (156)
3mm	Endian-swap a block of 16-bit values.	*ni: 400, nj:450, nk:500, nl: 550, nm: 600*	27 / (136)
atax	Endian-swap a block of 32-bit values.	*n: 10800, m: 10800*	31 / (73)
correlation*	Endian-swap a block of 64-bit values.	*n: 1000, m: 800*	39 / (174)
doitgen*	Move block of memory.	*nr: 120, n: 110, np:130*	13 / (62)
gemver*	Vector product of two input arrays.	*n: 10000*	7 / (67)
jacobi-2d*	Dot product of two arrays.	*n: 650, tsteps: 250*	17 / (175)
mvt	Complex FIR.	*n: 6000*	24 / (29)
nussinov *	Least Mean Square Adaptive Filter.	*n: 1100*	17 / (57)
seidel-2d	Convert IEEE FP into Q.15 format.	*n: 800, tsteps: 200*	16 / (27)
syr2k	Matrix Multiply.	*n: 600, m:500*	19 / (44)
syrk	Transposes a matrix of 16-bit values.	*n: 800, m:700*	8 / (43)

overhead of the operating system and/or other background programs may influence the quality of the measurements.

Other aspects that makes it important to measure multiple times, is the frequency/voltage scaling, which is difficult to predict as it can be influenced by a number of different variables (temperature, CPU/core use). For instance, Turbo Boost (Intel Corporation) increases the frequency and voltage on some cores of the Intel CPUs in order to accelerate applications that just use a few of the available cores. In the case of the ARM *big.LITTLE* architecture (ARM Limited, 2013), as a measure to save energy (the smaller cores are more energy efficient), the CPU migrates threads between the *big* and the *LITTLE* cores depending on the CPU workload.

Performance on Intel Xeon and ODROID-XU+E. For performance measurements we rely on `clock_gettime()` Linux calls, which compared with other calls for timing measurement (e.g., `gettimeofday()`) allows higher precision and the ability to request specific clocks. We use the the `CLOCK_MONOTONIC` clock. The Linux function `clock_getres()` reports one nanosecond of resolution in both platforms for the clock used.

Energy on Intel Xeon. For measuring energy on the Intel system we use the Running Average Power Limit (RAPL) interface (Rotem et al., 2012), which provides access to a mechanism to regulate power usage and to a set of registers with power and energy measurements. Power measurements are generated on the fly, per socket, using an on-chip energy model that relies on hardware performance counters and I/O. RAPL is available in Intel CPUs starting with the Sandy Bridge microarchitecture, and has been shown to correlate well with real measurements or at least to be indicative of overall energy and power trends (Rotem et al., 2012; Hackenberg

et al., 2015). RAPL gives access to four domains, *Package*, which includes CPU cores, memory cache, memory controller, *PP0* (CPU cores), *PP1* (GPU) and *DRAM*. For our experiments we evaluate energy consumption as the sum of the energy consumption reported for the *Package* and the *DRAM* domains of each CPU. We access the Intel RAPL measurements through the Linux *perf_event* subsystem. Although RAPL has been found to underestimate DRAM energy use on some architectures, this does not impact Haswell-EP CPUs, as this architecture uses a different estimation mechanism that includes actual measurements (Hackenberg et al., 2015). Haswell-EP does not support PP0 (Hackenberg et al., 2015), but that makes no difference for these experiments because we want to take into account the energy used by cache and the memory controller in addition to the energy used by the CPU cores. RAPL readings have a sampling frequency of close to one millisecond (1 KHz).

Energy on ODROID XU+E. The Odroid XU+E single board computer (Hardkernel) features 4 energy sensors: ARM, KFC, MEM and G3D. We query these sensors for energy measurements (except the G3D sensor), which refer to the A15 (big CPU), A7 (LITTLE CPU), memory and GPU subsystems, respectively. For each of these, voltage, current and power values are reported. The update period of the energy sensors of the Odroid XU+E devices is 264 milliseconds. We perform the power measurements for the ODROID by querying the `/dev/sensor_*` device files, using the `ioctl` function.

A.2.4 Datasets

PolyBench provides *mini*, *small*, *medium*, *large*, and *extra large* datasets. We selected one of these datasets on a function-by-function basis.

For each function, we selected/costumized a dataset to make the execution time of each function (compiled with –O3) larger than 1 second on the ODROID XU+E board in order to have enough samples to obtain sufficiently precise energy measurements.

We use the same datasets (i.e., the ones we selected on the ODROID platform) with the Intel Xeon platform, resulting in execution times much smaller than on the ARM-based board, but still well above what is required for precise energy measurements with RAPL (David et al., 2010).

A.2.5 Compilation and validation

We relied on the same compilation flow described in Section 3.4.2. Additionally, as we compiled and executed the application on the same platform, we used the *-mcpu=native* flag with the *llc* tool. This resulted in the generation of assembly code with vector instructions, NEON instructions for the ARM CPUs and AVX instructions for the Xeon CPUs. In addition, we used *-fp-contract=fast* with *llc* so that fused multiply-add machine instructions (supported by both the Intel and the ARM CPUs used) are used when possible.

In order to deal with a situation where the compiler tools halt (i.e., gets trapped in an infinite loop, for some combination of function, phase order and/or target), our DSE framework allows setting a time limit for each call to the tools (in this case, *clang*, *opt*, and *llc*).

For the validation step we always use the smallest available PolyBench/C dataset (i.e., the *mini* dataset), in order to reduce the execution time needed for the validation.

A.2.6 Compiler phase orders exploration

We generated a set of sequences by randomly generating 1,000 compiler sequences composed of 128 compiler passes each using our compiler phase selection/ordering exploration framework, using an ARC4-based pseudo-random number generator from (Bau, 2016). For each position of a compiler sequence, we randomly selected a pass from the set of LLVM passes presented in Section 5.3.2.

This set of sequences was iteratively used to compile each of of the functions considered, resulting in the generation of up to $2,000$ binaries for each function considered in the experiments. $1,000$ binaries for the OpenMP versions and $1,000$ binaries for the versions without OpenMP. We note that some sequences failed to produce valid binaries when compiling some of the functions. Energy consumption and execution time metrics were extracted from a single execution of each of those binaries, relying on the *OMP_NUM_THREADS* environment variable to set the number of OpenMP threads for the OpenMP versions.

Then, for any given function, we built a graph as in Section 4.2.2 with the phase orders (generated in the previous step) for all other 11 PolyBench/C functions (one phase order per function) for the configuration (i.e., no OpenMP, or OpenMP with specific number of threads) that resulted in the lowest energy consumption when compiling with the standard optimization levels. This model represents a sequence space from which not only the sequences used in its construction can be regenerated, but also many others sharing similarities with them. We used the leave-one-out approach to validate, implying the construction of 12 different models per target platform by considering the best phase orders for each of the functions except the one being optimized at any given time.

We relied each of the models to iteratively generate $1,000$*sequences* (sequences generated between different models differ), and used them in the compilation of the function correspondent to the use of each specific model.

Energy consumption and performance metrics were extracted in two steps. First, we evaluated each binary once. We then selected the 5% (i.e, 50) compiler phase orders that resulted in the binaries with lowest energy consumption. Finally, we executed the binaries generated with those phase orders for 25 times (the same number of times as for the –Ox experiments), and analyzed the energy consumption and performance of the binaries.

We identified, for each function and target platform (i.e., dual Xeon and ODROID), the –OX flags that leads to the lowest energy consumption for each number of threads on the OpenMP versions and for the compilation without OpenMP. Then we calculated, for each of the 50 compiler sequences previously selected, energy consumption and performance improvement ratios over

Table A.2: Best compiler flag and execution parameters for each kernel and target platform.

	Dual Xeon			ODROID		
Function	**Energy**	**Perf.**	**Power**	**Energy**	**Perf.**	**Power**
2mm	**16T – O3**	16T – O1	S – O3	**S – O2**	4T – O1	1T – O3
3mm	**16T – O3**	16T – O3	S – O2	**S – O3**	4T – O2	S – O3
atax	**16T – O3**	32T – O1	1T – O3	**S – O3**	4T – O1	S – O3
correlation	**S – O3**	16T – O1	S – O3	**S – O1**	S – O1	S – O0
doitgen	**S – O3**	S – O3	S – O3	**S – O1**	S – O1	1T – O2
gemver	**32T – O3**	32T – O3	1T – O3	**S – O3**	4T – O1	S – O1
jacobi-2d	**8T – O1**	8T – O2	S – O1	**S – O2**	4T – O2	S – O0
mvt	**16T – O2**	16T – O3	1T – O1	**S – O2**	4T – O2	1T – O1
nussinov	**16T – O1**	16T – O1	S – O3	**S – O3**	4T – O1	S – O1
seidel-2d	**16T – O3**	32T – O2	S – O1	**2T – O2**	4T – O0	S – O1
syr2k	**16T – O2**	16T – O2	S – O2	**S – O2**	4T – O3	S – O2
syrk	**16T – O1**	32T – O3	S – O1	**S – O3**	S – O2	S – O1

those –OX configurations. For instance, for the *2mm* function on the dual Xeon workstation, the lowest energy consumption version is the one using OpenMP with 16 threads (as with most other functions on the Xeon target) and with the –O3 optimization level; therefore ratios presented here. For each function and target pair, we report the energy consumption and performance improvement ratios for the version using the same number of OpenMP threds (or without OpenMP).

A.3 Results

We first present results of energy consumption, performance and average power for the functions compiled with the default optimization flags. Including results for the kernels compiled without OpenMP support, and with OpenMP support considering different numbers of OpenMP threads.

In addition, we present results for exploration of compiler phase orders focusing on energy consumption, generated with the methodology explained in Section A.2.6. We analyze those results by comparing them with the energy consumption and performance of the binaries generated with the standard optimization level flags, and we also individually comment on the level of correlation between energy consumption and performance.

Energy and performance value pairs are obtained by averaging 25 executions of each of the binaries resulting from compilation of a given function with a specialized phase order (or a –Ox flag).

Table A.2 presents the compilation (i.e., which optimization flag) and execution configurations that lead to the lowest energy consumption, the best performance, or the lowest average power consumption. Serial execution is represented by *S* for the version compiled without OpenMP and *1T* (i.e., OpenMP with a single thread) otherwise.

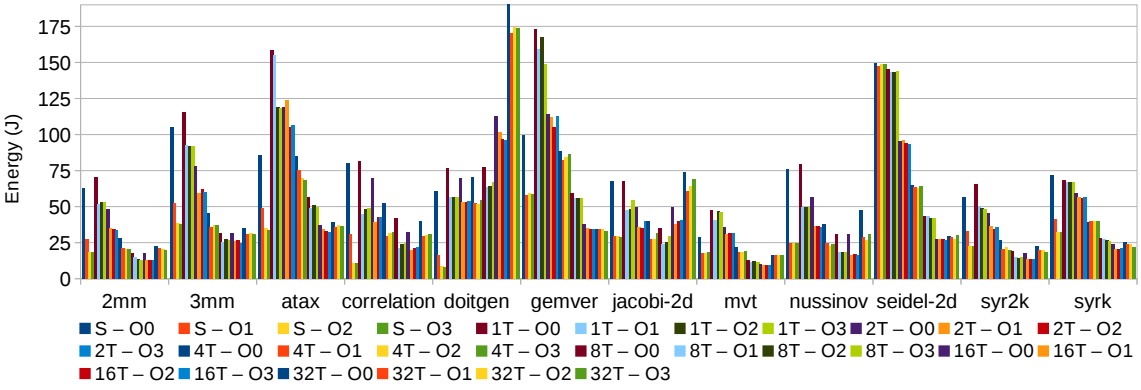

Figure A.1: Energy consumption in joules for each function when targeting the Dual Xeon with the standard optimization flags.

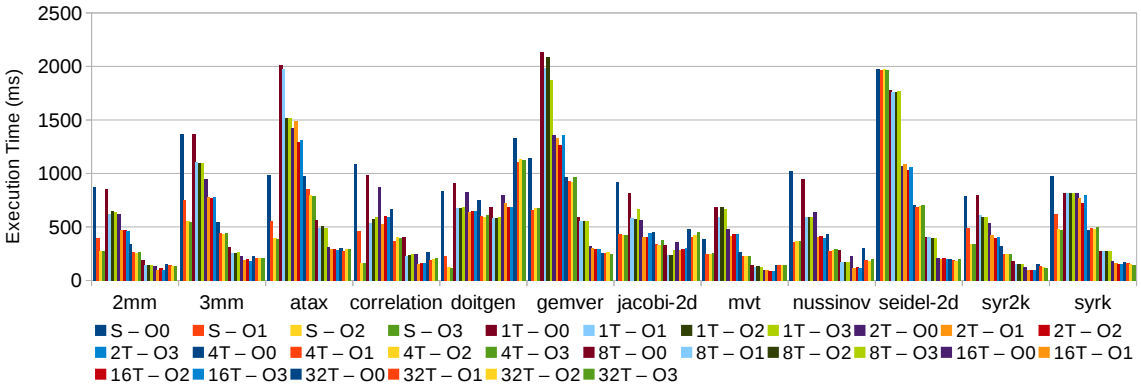

Figure A.2: Execution time in milliseconds for each function when targeting the Dual Xeon with the standard optimization flags.

A.3.1 Energy and performance with standard optimization levels

We present results for the use of 1, 2, 4, 8, 16 and 32 OpenMP threads on the dual Xeon and 1, 2 and 4 threads on the ODROID when executing the functions compiled with the standard optimization flags, as well as results for experiments without OpenMP.

A.3.1.1 Dual Xeon

Figures A.1, A.2 and A.3 depict the absolute energy consumption (in joules), the absolute performance (in milliseconds), and the average power consumption (in watts), on the dual socket Xeon platform for the execution with and without OpenMP of the 12 PolyBench/C functions considered for the experiments. The functions were compiled with no optimization and with the –O1, –O2, and –O3 standard optimization flags. Figure A.4 represents energy consumption and execution time on the same chart for the binaries generated by compilation of the considered functions with the standard optimization levels. Considering a number of execution threads, the use of the standard optimization levels (i.e., –O1, –O2, –O3) always results in the generation of binaries with both the lowest energy consumption and highest performance, when compared with the binaries

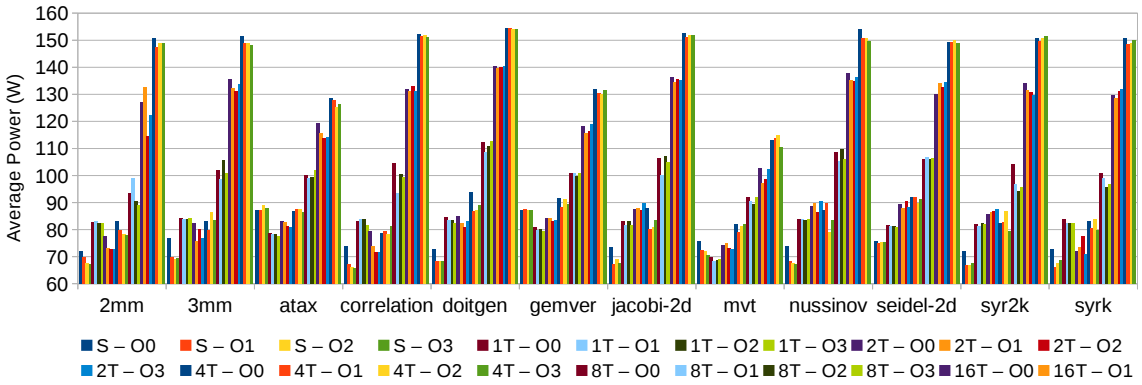

Figure A.3: Average power in watts for each function when targeting the Dual Xeon with the standard optimization flags. Calculated from energy consumption and execution time by $P = E/\Delta t$.

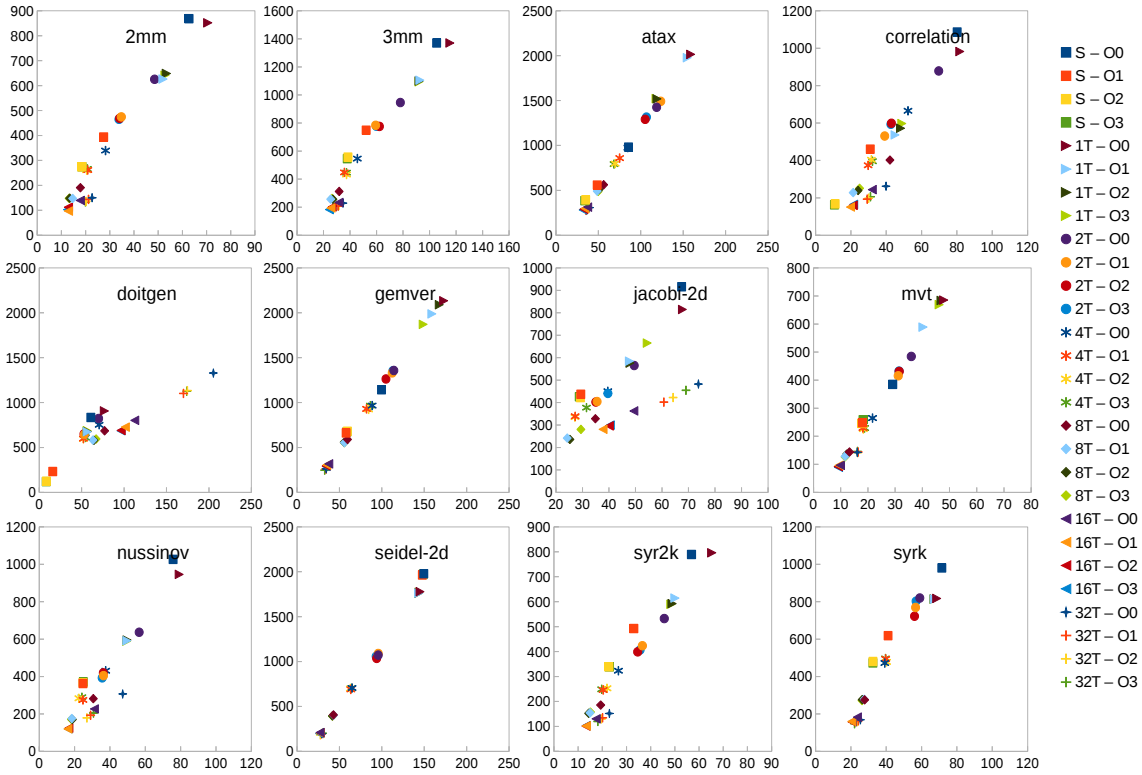

Figure A.4: Energy consumption in joules (horizontal axis) and execution time in milliseconds (vertical axis) on the Dual Xeon.

generated without optimization. The improvement obtained by using those flags is considerable, especially when OpenMP is not used (up to 87% energy consumption reduction). Additionally, when not using OpenMP –O2 and –O3 tend to improve energy consumption and performance over –O1. The only exception are *gemver* and *seidel*. The use of –O1 with *gemver* results in saving 42% energy (vs. not using any optimization level), but using –O2 or –O3 results in no additional improvement. With *seidel-2d* none of the optimization levels resulted in saving energy

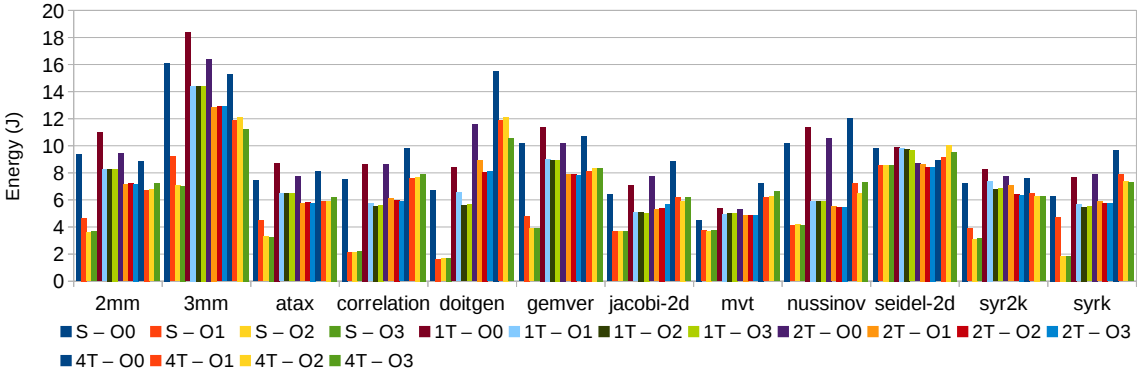

Figure A.5: Energy consumption in joules for each function when targeting the ODROID with the standard optimization flags.

or improving performance. When OpenMP is used the norm is that –O2 and –O3 do not tend to reduce the energy consumption or the performance further over –O1 by a significant amount, and in fact sometimes result in less energy efficient binaries. The binaries optimized with those optimization levels also tend to have lower average power consumption (*atax*, *gemver*, *seidel-2d* are the exceptions) if OpenMP is not used. The use of OpenMP resulted in performance degradation for some functions when using less than two threads. Three of the most extreme cases were the *atax*, *correlation* and *doitgen* functions. The *atax* and the *correlation* functions need 16 threads for the OpenMP version to surpass the versions without OpenMP. Interestingly, the *correlation* function without OpenMP (with –O2 or –O3) consumes 2× less energy than any other version, making this a good example to show that energy and execution time are not always correlated. The serial version without OpenMP of the *doitgen* function is faster than any OpenMP version, independently on the number of threads used. The use of 16 and especially 32 threads considerably negatively impacts energy consumption more than performance. Other example where the use of more threads (16 or 32 threads) hurts energy consumption and performance disproportionally is *jacobi-2d*. Performance tends to improve over the serial non-OpenMP with the use of 4 or 8 OpenMP threads. As expected, the average power tends to increase with the number of OpenMP threads. The inverse behavior is seen for some functions when going from 1 to 2 OpenMP threads (*2mm*, *3mm*, *correlation*, *syrk*) or from 2 to 4 threads (*jacobi-2d*, *nussinov*, *syr2k*) The serial versions without OpenMP tend to use less power, being *atax*, *gemver*, *mvt* the exceptions.

A.3.1.2 ODROID XU+E

Figures A.5, A.6 and A.7 depict energy consumption, performance and power on the ODROID. Figure A.8 shows energy consumption and execution time on the same chart for the binaries generated by compilation of the considered functions with the standard optimization levels. As with the Xeon-based platform, for the same function configuration (i.e., without OpenMP or with a given number of OpenMP threads), on the ODROID the use of –O1, –O2 or –O3 tends to result in lower energy consumption (the non-optimized versions never consumed less energy). The only

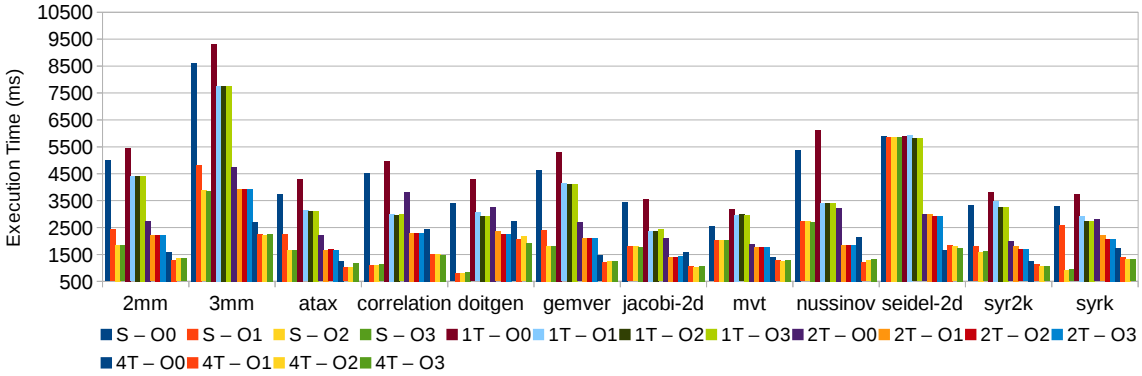

Figure A.6: Execution time in milliseconds for each function when targeting the ODROID with the standard optimization flags.

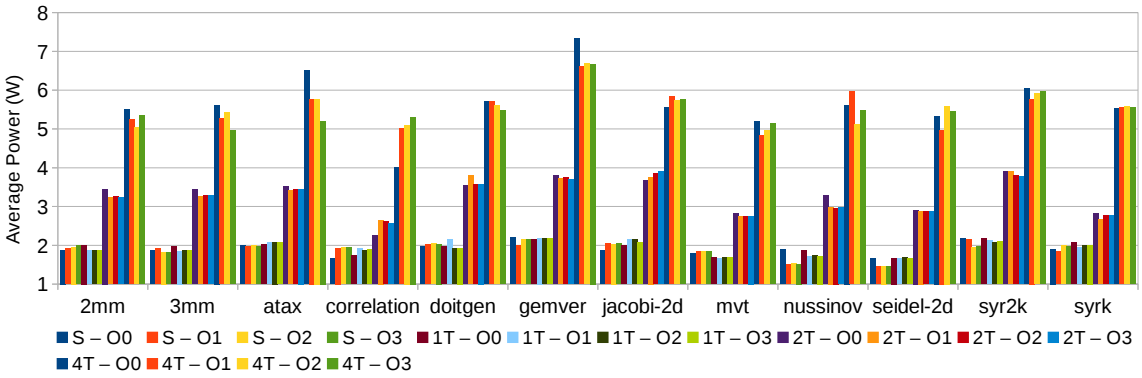

Figure A.7: Average power in watts for each function when targeting the ODROID with the standard optimization flags.

exception is *seidel-2d* with 4 OpenMP threads, in which case energy consumption increases with the use of any optimization level. Performance and energy consumption and optimization levels are closely related. If energy consumption decreases, then wall time tends to decrease. These changes do not always happen in the same proportion for both metrics. For instance, *seidel-2d* without OpenMP consumes up to 13% less energy if compiled with the standard optimization levels, while improving less than 1% in terms of performance in comparison with not relying on any of the optimization levels. Compilation without OpenMP resulted in the generation of binaries that use less energy, with the *seidel-2d* function being the only exception. For this function the binary compiled with OpenMP saves more energy if executed with two threads. The ODROID consumes less energy than the dual Xeon workstation and the dual Xeon based system executes the compiled functions much faster than the ARM processors on the ODROID. The average power consumption on the ODROID is also much lower, with all functions executing bellow 8 watts.

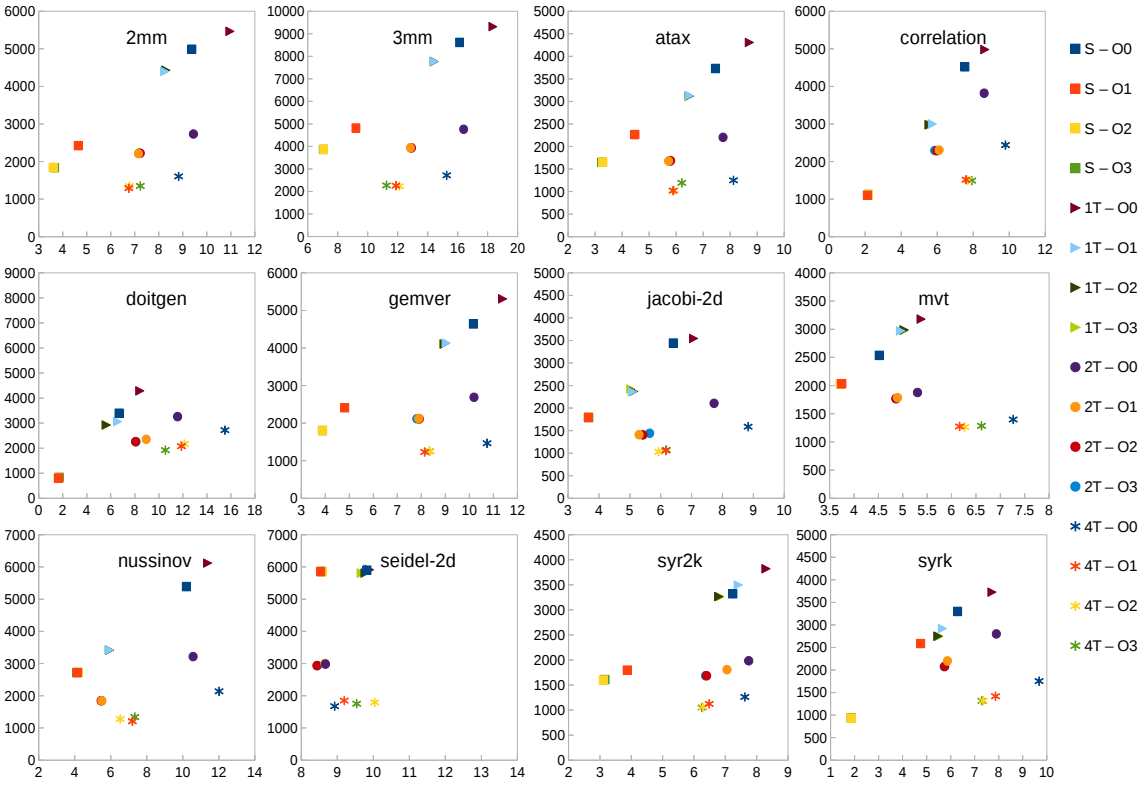

Figure A.8: Energy consumption in joules (horizontal axis) vs. execution time in milliseconds (vertical axis) on the ODROID.

A.3.2 Energy and performance with the generated optimization sequences

We present next plots of energy/performance improvement ratios for both the dual Xeon and the ODROID platforms. The ratios are calculated for the use of the generated compiler phase orders over the best compilation/execution configurations for energy, as depicted in Table A.2. For instance, for *gemver* on the Xeon, the best configuration uses –O3 with 32 threads, so the results that we show next are ratios over that configuration for the use of the generated phase orders when compiling with OpenMP and executing with 32 threads. Results falling over a straight line represent compiler phase orders that resulted in directly correlated energy consumption and performance (increases in energy efficiency lead to proportional increases in performance, and vice versa). Points diverging to one side or the other represent compiler sequences that lead to non-aligned energy efficiency or performance gains.

A.3.2.1 Dual Xeon

Figure A.9 shows energy consumption and execution time ratios for the binaries generated with phased orders specialized for improving energy efficiency. The results presented are ratios over the best individually found –OX (i.e., –O0, –O1, –O2, or –O3) for each function, metric and target/configuration triplet. The data collected from the experiments suggests that although energy

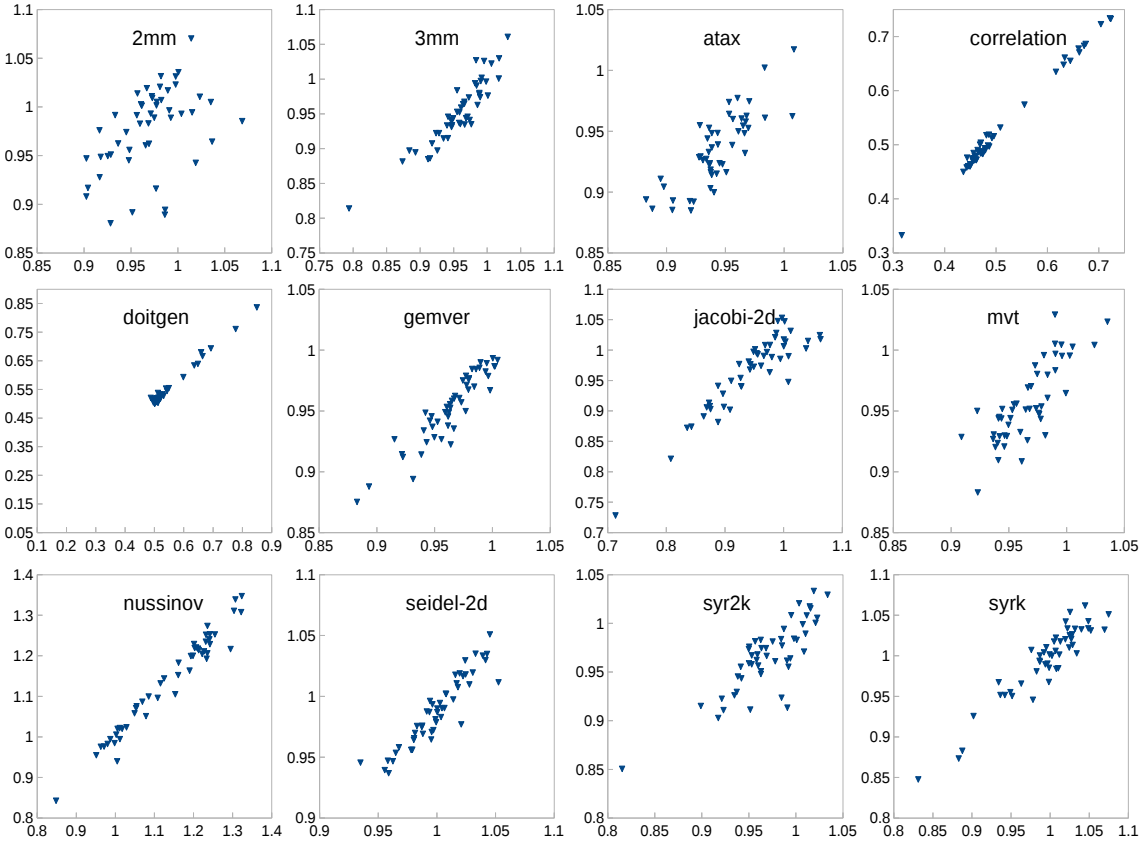

Figure A.9: Ratios of energy consumption (horizontal axis) and execution time (vertical axis) over the best per-function standard optimization level and execution configuration on the dual Xeon.

efficiency and performance tend to correlate, there are sequences that lead to cases of performance gains without energy improvements, or vice versa. For instance, in the case of *2mm*, there are a lot of energy/performance ratios all over the chart, meaning there are a lot of sequences resulting in asymmetrical energy reduction and performance gains. Additionally, for *2mm* the compiler sequence that leads to best energy savings (point most to right) does not lead to the highest performance (point most to the top). For *atax* two compiler phase orders result in the same best energy efficiency (the two points most at right) while having different performance. For the functions where the single threaded non-OpenMP version resulted in best energy efficiency, such as *correlatio, doitgen*, the points on the chart form a straight line.

A.3.2.2 ODROID XU+E

As with the Xeon platform, although energy consumption and performance are for most part highly correlated (i.e., both tend to improve or get worse together), there are pairs of points (representing ratios of energy efficiency and performance using specialized phase orders) for any functions where energy decreases without performance increasing at all, and vice versa. For instance, with *2mm*, *doitgen*, *jacobi-2d*, *nussinov*, *mvt* and *syrk* there are a number of sequences that result in considerably more energy reduction than impact on performance.

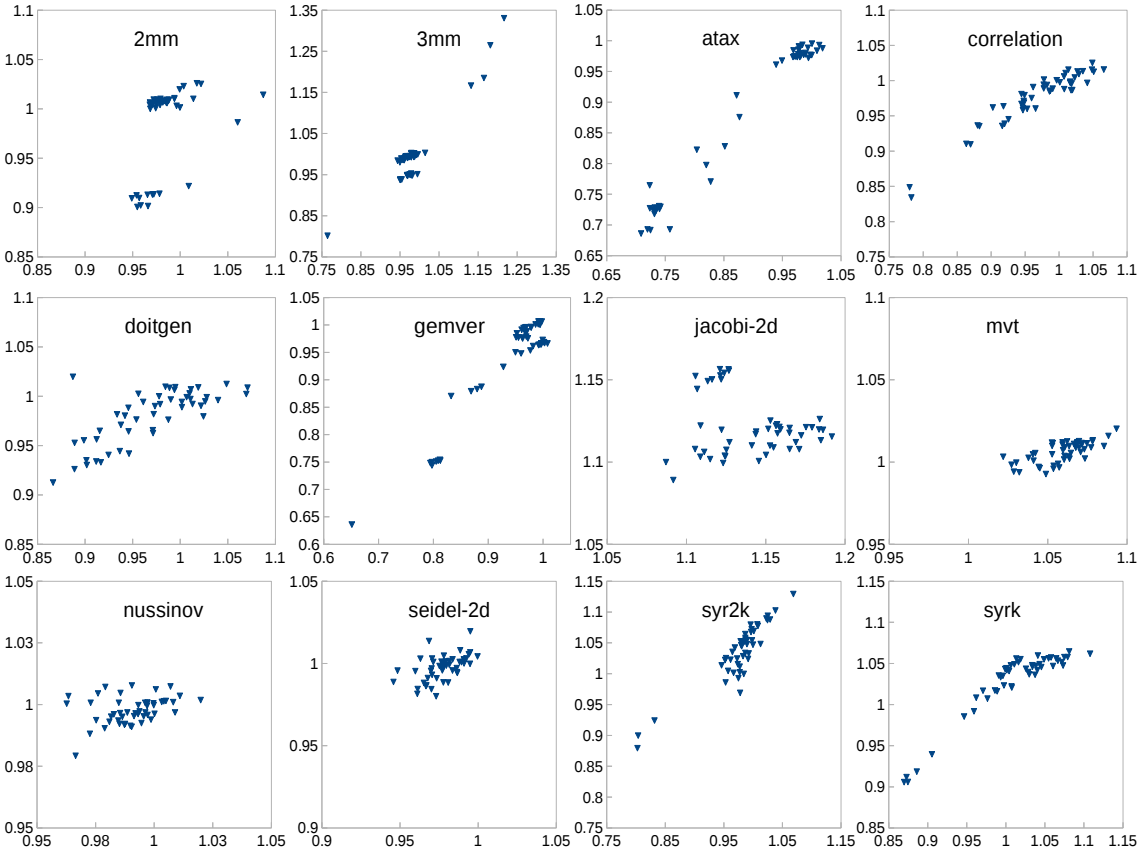

Figure A.10: Ratios of energy consumption (horizontal axis) and execution time (vertical axis) over the best per-function standard optimization level and execution configuration on the ODROID.

Appendix B

Source-to-Source Transformation + Direct-Driven Approach targeting Multicore Architectures

In the context of controlling source-to-source tools, LARA allows transforming unannotated program code into code decorated with target-specific directives, and optimized for the target architecture. In this approach, programmers specify source-to-source transformation actions as LARA aspects. Each aspect defines the application of a sequence of transformations (a strategy). Aspects can be composed to form sophisticated transformations exploiting specific features of the target system. With this approach, programmers can easily experiment different sequences of transformations, different target directive systems and the corresponding compilers. As programmers do not directly modify the original source code, this approach has the benefit of promoting program portability.

Figure B.1 presents an aspect with the goal of inserting OpenACC pragmas in critical loops. The select statement identifies every loop in the program whereas the apply section instructs the weaver to perform an action to each of the identified loop, if the expression in the condition holds. This condition states that only loops with an estimated cost greater than fraction will be transformed.

The importance of LARA when targeting parallel devices using directive-driven models derives from the fact that effective heuristics for GPU targets, (e.g., (Magni et al., 2013)) can be encapsulated in a LARA DSE scheme. The LARA interpreter can execute all the needed tools (e.g., profilers, source-to-source compilers) in the required order and with input parameters determined by a DSE scheme.

We present in the next sections Source-to-Source transformations guided by a LARA DSE scheme in the context of compilation targeting CPUs and GPUs. Our approach is able to efficiently instrument code in a way that different multiple versions of a given function are generated and tested. The experiments using a matrix multiplication kernel show that the best approach in terms

```
aspectdef optimize
   input fraction end
   select function.loop end
   apply $loop.insert before '"#pragma acc kernels';   end
   condition $loop.estimated_cost > fraction   end
end
```

Figure B.1: Aspect mapping loops with a considerable percentage of the execution time to an accelerator.

of what is the best set of source-to-source code transformations to perform is very dependent on the target architecture, even if only considering targets of the same type (e.g., GPUs).

B.1 LARA-based source-to-source transformations using MANET

MANET (Pinto et al., 2015) performs a series of steps in order to transform the input code in a way it reflects the instructions captured by LARA aspects. The LARA compiler (larac) receives join point and attribute models for the C language and the aspect files describing the weaving actions to be performed. With these, larac generates a file with an intermediate representation of the aspect, Aspect-IR. Next, larai interprets the Aspect-IR and communicates with the weaving engine. Finally, the weaver directs our Cetus (Lee et al., 2004; Dave et al., 2009; Bae et al., 2013) instance for weaving actions over the input C code files. The flow also includes the possibility of feedback allowing information to be sent back to the interpreter, e.g., join points or join point attributes. This provides an iterative flow where requests and actions go from larai to Cetus and back to larai, via the weaving engine on both passes. Figure B.2(a) shows this flow and how the different components interact.

Currently, the weaver supports four loop-based code transformations, namely: Loop Tiling, Loop Interchange, Loop Unroll, and Unroll-and-Jam. The unroll-based loop transformations require the selection of a loop followed by the invocation of the corresponding action providing an unroll factor.

The aspects related to LARA DSE strategies can include instructions to execute tools, explore configurations and/or command line options, get attribute values from reports, and decide to continue exploring different configurations/options based on the results achieved at a particular stage of the toolchain. Hence, LARA aspects can implement DSE schemes, relying on this outer-loop mechanism as depicted in Figure B.2(b).

The DSE loop, described by an external aspect, is responsible for feeding the LARA aspect presented in Figure B.2(b) with the exploration parameters, which are input to the aspects. That aspect, alongside with the C code, are input to MANET, which outputs C source files annotated with compiler-specific pragmas. After the compilation of this code using backend tools and its execution/simulation, the DSE loop extracts information from the execution reports, which is then used in the process of deciding the best design parameters. The loop execution resumes as new

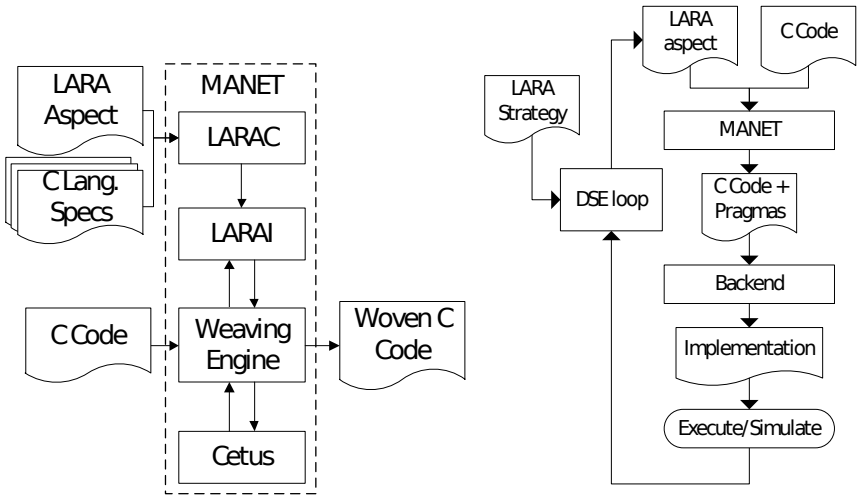

Figure B.2: MANET flow (a) and a possible DSE scheme toolchain (b). Reprinted from (Nobre et al., 2013b).

```
aspectdef DSE_loop
   var best_fraction, best_latency = INF;
   for(var fraction = 100; fraction >=  10; fraction--) {
       run('manet', ['optimize.lara', '-av', fraction]);
       report('accULL');
       if(@latency < best_latency) {
          best_latency = @latency; best_fraction = fraction;
       }
   }
   print(``bFrac='' + best_fraction + ``, bLat = '' + best_latency);
end
```

Figure B.3: A DSE loop for exploring the parallelization of loops using more or less CPU time than a given "fraction" value.

values are fed to the LARA aspect. Figure B.3 depicts a LARA aspect that implements this DSE loop (note that this example is very simple and we can use very elaborated DSE strategies). The aspect uses a loop to control the value of the variable fraction, the DSE parameter. This value is used as an argument when we use run to weave the code using MANET with the aspect optimize, shown in Figure B.1. Then, using report, we run our backend tools to compile and execute the binaries and capture this execution's results. Finally, it is possible to test the execution time values resulting from the execution and compare them with the previously saved best value.

B.2 Experiments

We now describe a set of experiments aimed at evaluating the effectiveness of LARA aspects as a mechanism to specify source code transformations when exploiting concurrency. In these exper-

```
for(i=0; I < n; i++)
   for(j=0; j < p; j++)
      for(k=0; k < m; k++)
         C[i*p + j] += A[i*m + k] * B[k*p + j];
```

Figure B.4: Triple-nested loop implementation of GEMM matrix multiplication algorithm.

iments we focused on a specific code implementation of the General Matrix Multiply (GEMM), a widely used computational kernel in science and engineering applications. We began with a sequential C version of the computation (depicted in Figure B.4) and used LARA aspects and MANET to automatically generate a series of transformed code versions of this computation. These versions include concurrency specification constructs via OpenMP/OpenACC pragma annotations as well as basic and advanced core transformations for enhanced performance.

The developed LARA aspects insert pragmas targeting directive-driven compilation models/-tools, as well as instruct compiler tools, specifically Cetus, to perform code transformations/optimizations at specific points of interest in the source code.

B.2.1 Design exploration using LARA

We present an iterative exploration of compiler optimizations on the GEMM kernel depicted in Figure B.4. LARA aspects were developed to incrementally transform this code thus allowing us to evaluate in an incremental fashion the performance benefits of different code optimizations.

For each transformed version of the kernel, two baseline implementations were generated, using single-precision and double-precision floating-point data representations, respectively. In addition to these baseline versions, we also explored four different code versions targeting a traditional CPU architecture, by applying the following transformations (specified in LARA):

1. $LT_{i,j,k=32}$: Loop tiling on all loops, implementing what is known as the blocked/tiled matrix multiplication.

2. $LT_{i,j,k=32} + LI_{j,k}$: Loop tiling and loop interchange of the two innermost loops.

3. $LT_{j,k}$: Loop interchange of the two innermost loops.

4. $LT_{j,k} + UJ_i$: Loop interchange on the two innermost loops and unroll-and-jam on loop `i`.

OpenACC pragmas were also inserted to exploit whenever possible available concurrency when targeting the GPUs, resulting in the source code depicted in Figure B.5 for a non-optimized GEMM kernel. Unlike commercial OpenACC compilers (e.g., from PGI and CAPS), the compiler we used for our experiments, accULL (Reyes et al., 2012), does not perform a number of optimizations (e.g., loop unrolling, loop tiling). To compensate for this lack of features, we relied on a set of LARA aspects that carried out these optimizations using MANET (Pinto et al., 2015).

```
#pragma acc kernels copyin(A[0:n*m],B[0:m*p]) copy(C[0:n*p]) {
   #pragma acc loop private(i) gang independent
   for(i=0; i<n; i++)
      #pragma acc loop private(j) worker independent
         for(j=0; j<p; j++)
            for(k=0; k<m;  k++)
               C[i*p + j] += A[i*m + k] * B[k*p + j];
}
```

Figure B.5: GEMM annotated with OpenACC pragmas.

Starting with the code version depicted in Figure B.4, three new code versions targeting GPUs were incrementally generated (using LARA aspects), by directing MANET to apply the following transformations:

1. (I0) Original version with OpenACC pragmas inserted as depicted in Figure B.5.
2. (I1) $LI_{i,j}$: Loop interchange of the two outermost loops.
3. (I2) $LI_{i,j}$ + $LINV_k$: Loop interchange of the two outermost loops and loop invariant code motion on the memory read and write from/to the output matrix.
4. (I3) $LI_{i,j}$ + $LINV_k$ + $LT_{i,j=4}$: Loop interchange of the two outer-most loops, loop invariant code motion on the memory read and write from/to the output matrix, and loop tiling of the two outermost loops using a tiling factor of four.

B.2.2 Target platforms

We carried out performance evaluation experiments on four hardware platforms with the following configurations:

1. **Target 1.** PC with two Intel Xeon 5050 dual-core processors clocked at 3.0 GHz with 2 MB of L2 cache per core and 4 GB of DDR2-667 SDRAM.
2. **Target 2.** PC with two IBM PowerPC (PPC) 970 processors at 1.8 GHz, with 512 KB of L2 cache per processor and 1 GB of DDR-400 SDRAM.
3. **Target 3.** PC with an Intel Core i5 2500K quad-core processor at 3.3 GHz with 6 MB of L2 cache and 4 GB of DDR3-1333 SDRAM, paired with two NVIDIA GeForce 400 Series GPUs: a GTX 460 and a GTX 480.
4. **Target 4.** Laptop with an Intel Core i7-3630QM (3.4GHz Turbo) quad-core processor at 2.4 GHz with 8GB of DDR3-800 SDRAM and a NVIDIA GeForce GT 650M.

Target 1 uses SUSE Linux Enterprise Server 10 64-bit with GCC 4.5.2. Target 2 uses Debian 5.0 64-bit with GCC 4.3.2. Target 3 and 4 use the Ubuntu 12.04 64-bit distribution with GCC

Table B.1: Target GPU architectures.

GPU	Core/Shader Clock	VRAM / Bus	SPs	FP64
GTX 460 (GF104)	675/1350 MHz	1 GB GDDR5 @ 3600 MHz / 256-bit	336	1/12 FP32
GTX 480 (GF100)	700/1401 MHz	1.5 GB GDDR5 @3696 MHz / 384-bit	480	1/8 FP32
650M (GK107)	850 MHz	4 GB DDR3 @ 1800 MHz / 128-bit	384	1/24 FP32

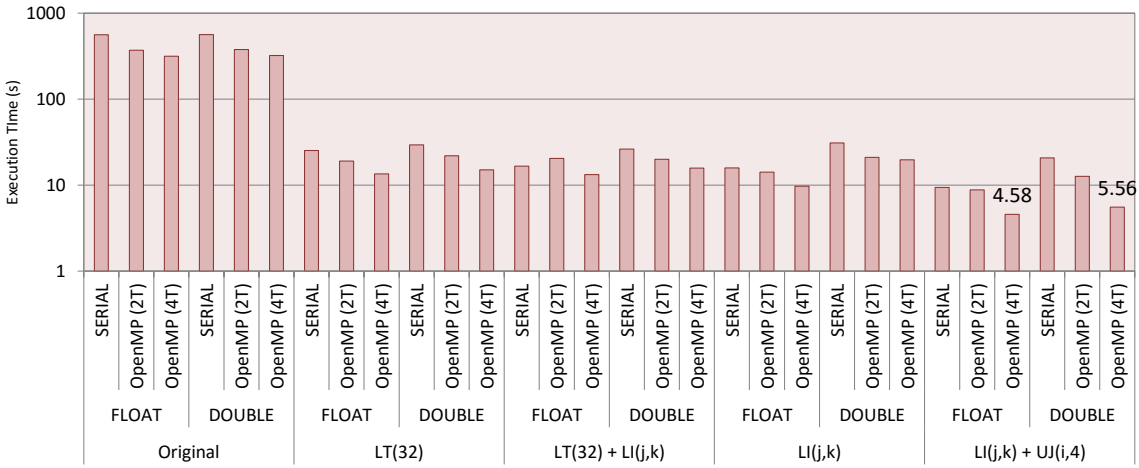

Figure B.6: Execution time (in secs) of CPU GEMM serial and OpenMP implementations (2 or 4 threads) on Target 1 (Intel Dual Xeon 5050).

4.7.2, CUDA 5.0, and the NVIDIA 304.88 and NVIDIA 310.14 drivers, respectively. The three GPUs (see Table B.1) support fused multiply-add single-precision floating-point operations.

We use the GCC compilers with the -O2 optimization level and the `-ftree-vectorize` flag for all targets, and the `-fopenmp` flag for the OpenMP implementations. Additionally, the `-march=native` and `-mcpu=970` were used when compiling respectively for the x86-64 and PPC targets to generate assembly code that uses all features of the target processors. CUDA kernels were compiled with the NVIDIA CUDA Compiler (NVCC) using the `-arch=sm_20` flag.

We now present performance results of each implementation/target pair for squared matrix shapes of 2048×2048 values. All reported results are averages of multiple executions.

B.3 Experimental results

Figure B.6 depicts the performance results (in sec) for the code versions resulting from applying the CPU-oriented optimizations described above for the execution on Target 1. Figure B.7 shows the performance results for Target 2.

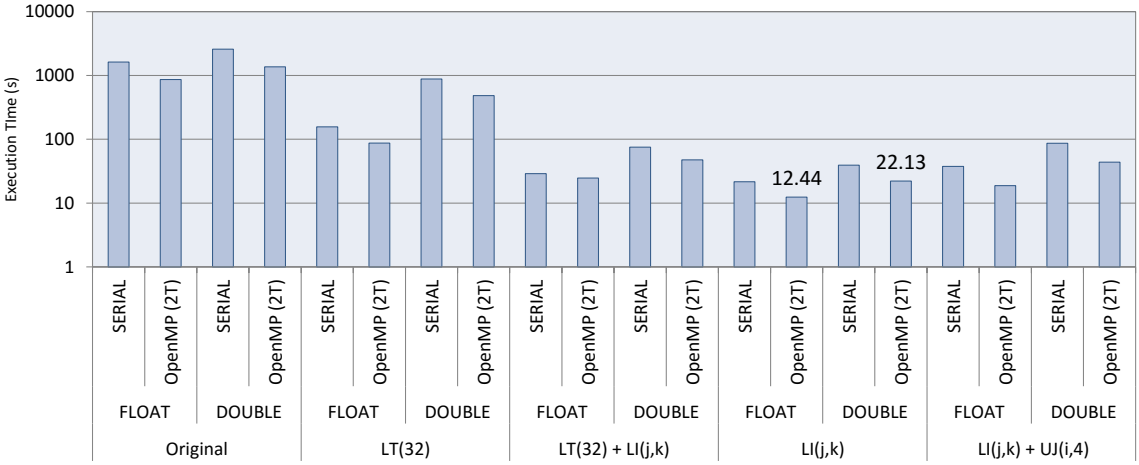

Figure B.7: Execution time (in secs) of CPU GEMM serial and OpenMP implementations (2 threads) on Target 2 (Dual IBM PowerPC 970).

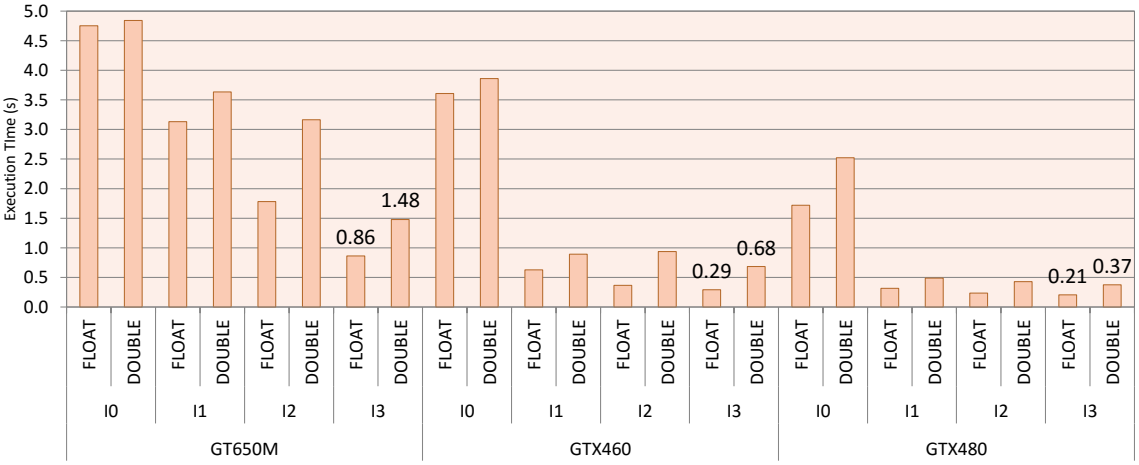

Figure B.8: Execution time (in secs) of CUDA GEMM implementations on Target 3 (NVIDIA 650M) and Target 4 (NVIDIA GTX460 and GTX480).

Lastly, Figure B.8 depicts the performance results when executing CUDA versions generated by accULL on the NVIDIA GPUs (Targets 3 and 4).

B.3.1 Discussion of results targeting the CPUs

As expected, the performance of the original GEMM kernel exhibits similar performance for both single- and double-precision versions. The blocked GEMM implementation, which resulted from applying the loop tiling transformation on top of the original GEMM, achieved much better performance both on the PowerPC and the Xeon-based targets, as a 32×32 data block completely fits in L2 cache, while being large enough to "considerably" improve locality. While the same tile size was independently selected for both processors using the same LARA DSE scheme, more considerable performance improvements were measured on the Intel Xeon processor than on the

PowerPC. Applying loop interchange to the two innermost loops of the triple-nested loop block, instead of applying loop tiling, resulted in even better performance on both the PowerPC and the Intel Xeon when considering single-precision floating-point data. The performance increase is at least partially due to GCC's success in vectorizing the innermost loop, which include the multiply/add operations. On the Xeon processor, applying unroll-and-jam to the "i" loop resulted in a two-fold ($2\times$) performance gain when using single-precision and up to four-fold ($4\times$) when using doubles. On the PowerPC, the same transformation led to close to a two-fold slowdown both using single and double data types.

B.3.2 Discussion of results targeting the GPUs

The base GPU OpenACC code did not exhibit good performance on any of the tested GPU targets. The performance of single- and double-precision floating-point is very similar (e.g., 4.75/4.84 sec single/double on the 650M), due to the poor utilization of the GPU resources.

For these GPU versions, performing loop interchange over the two outermost loops (in contrast with the two innermost when targeting the CPUs) resulted in a substantial increase in performance as it significantly improved the corresponding data locality and hence the cache behavior. Loop interchange led to the most substantial speedup on both Fermi GPUs, with all measured speedups greater than $4\times$ on both single and double-precision. The performance gap between single and double precision is more noticeable for the GPU versions with loop interchange (e.g., 3.13/3.63 sec single/double for 2048×2048 matrices on the 650M). In addition, applying the loop invariant transformation to the innermost loop resulted in a performance increase close to $2\times$ for single-precision on the 650M, and a much lower speedup for double-precision. On the Fermi cards, loop invariant did not make a considerable difference in terms of performance, resulting even in slowdowns for the GTX 460. Mapping calculations in tiles of 4×4 values for the output matrix, instead of individually calculating the value for each adjacent `j` value on a different thread, resulted in better resource utilization on all GPUs, in particular on the 650M, where performance improved $2\times$ for both single and doubles.

Although the performance on all GPUs was better than on any CPU, it was very far from what is theoretically possible for matrix multiplication. For instance, the fastest single-precision implementation on the GTX 480 needed 0.21 sec to multiply two 2048×2048 matrices. Currently the CUDA/OpenCL kernels generated using accULL do not make use of the shared memory by stream processors. In addition, the difference between single and double-precision performance on the GEMM kernels was not nearly as high as 1/8 and 1/24, for the Fermi and Kepler GPUs, respectively, the difference announced by NVIDIA between single and double-precision performance. Furthermore, we are relying on the heuristics from accULL for the selection of the shape of gang/worker (i.e., block/thread) clauses. Fine-tuning the values for those clauses can potentially improve the performance of the generated implementations.

Appendix C

Phase Ordering Targeting GPUs

High Performance Computing (HPC) can offer Petaflops of performance by relying on increasingly more heterogeneous systems, such as the combination of Central Processing Units (CPUs) with accelerators in the form of Graphics Processing Units (GPUs) programmed with languages such as OpenCL (Khronos, 2015) or CUDA (Nickolls et al., 2008). Heterogeneous systems are widespread as a way to achieve energy efficiency and/or performance levels that are not achievable by a single device/architecture (e.g., matrix multiplication is much faster in GPU than CPU for the same power/energy budget). These accelerators offer a large number of specialized cores that the CPUs can use to offload computation that exhibits data-parallelism and often other types of parallelism as well (e.g., task-level parallelism). This adds an extra layer of complexity if one wants to target these systems efficiently, which in the case of HPC systems such as supercomputers is of utmost importance. An inefficient use of the hardware is amplified by the magnitude of such systems (hundreds/thousands of CPU cores and accelerators), with increasing utilization/power bill and/or cooling challenges as a consequence. Ensuring that the hardware is efficiently used is in part the responsibility of the compiler used (e.g., GCC and Clang) to target the code to the computing devices in such systems and also the responsibility of the compiler users. The programmer(s) and the compiler(s) have to be able to target different computing devices (CPU, GPU, and/or FPGA) and/or architectures (e.g., system with ARM and x86 CPUs) in a manner that achieves suitable results for certain metrics, such as execution time and energy efficiency.

Heterogeneous systems typically include a number of sub-devices with substantial differences. For this reason, different optimization strategies are needed for each computing component. With phase ordering, we can achieve closer-to-optimal optimization for these sub-devices, by specifying different compiler sequences for each of them. This approach is orthogonal to other optimization strategies. Additionally, HPC tend to prioritize metrics such as energy efficiency that typically receive less attention from the compiler developers, so these domains benefit can further from these specialized sequences.

The work presented here was performed with the following objectives:

1. Compare performance between OpenCL and CUDA kernels implementing the same freely available and representative benchmarks (PolyBench/GPU) using recent NVIDIA drivers

and CUDA toolchain, on an NVIDIA GPU with an up-to-date architecture (NVIDIA Pascal).

2. Assess the performance improvement that can be achieved using compiler pass phase ordering specialization with LLVM 3.9, in comparison with both use of that same LLVM compiler version without the use of phase ordering specialization and in comparison with the default OpenCL and CUDA kernel compilation strategies to NVIDIA GPUs[1].

3. Present additional experiments that help demonstrate the importance of specialization to a given OpenCL kernel and the importance of the order in which compiler passes are used.

4. Explain at the NVIDIA PTX assembly level (for OpenCL compilation with LLVM and for CUDA compilation with NVCC), for each kernel, what is causing the performance to improve with the use of specialized compiler sequences.

5. Experiment with the use of cosine similarity between vectors of code features to suggest compiler sequences.

C.1 Experimental setup

We extended our compiler phase ordering Design Space Exploration (DSE) framework to support exploring compiler sequences targeting NVIDIA GPUs using Clang/LLVM and the libclc OpenCL library.

We used a workstation with an Intel Xeon E5-1650 v4 CPU, running at 3.6 GHz (4.0 GHz Turbo) and 64 GB of Quad-channel ECC DDR4 @2133 MHz. For the experiments we relied on Ubuntu 16.04 64bit with the NVIDIA CUDA 8.0 toolchain (released in Sept. 28, 2016) and the NVIDIA 378.13 Linux Display Driver (released in Feb. 14, 2017).

The GPU used for the experiments is a variant of the NVIDIA GP104 GPU in the form of an EVGA NVIDIA GeForce GTX 1070 graphics card (08G-P4-6276-KR) with a 1607MHz/1797MHz base/boost graphics clock and 8GB of 256 bit GDDR5 memory with a transfer rate of 8008MHz (256.3 GB/s memory bandwidth). The graphics card is connected to a PCI-Express 3.0 16x interface.

The GPU is set to persistence mode with the command `nvidia-smi -i <target gpu> -pm ENABLE`. This forces the kernel mode driver to keep the GPU initialized at all instances, avoiding the overhead caused by triggering GPU initialization at application start. The preferred performance mode is set to *Prefer Maximum Performance* under the *PowerMizer settings* tab in the *NVIDIA X Server Settings*, in order to reduce the occurrence of extreme GPU and memory frequency variation during execution of the GPU kernels.

In order to reduce DSE overhead, and given the fact that we found experimentally that multiple executions of the same compiled kernel on the GTX1070 GPU had a small standard deviation in

[1] To the best of our knowledge this is the first work to present results of compiler pass phase ordering specialization targeting GPUs and considering OpenCL kernels.

relation to registered wall time, each generated code is only tested a single time during DSE. Only in a final phase on the DSE process are the top solutions executed 30 times and averaged. All execution time metrics reported (baseline CUDA/OpenCL and OpenCL optimized with phase ordering) correspond to the average over 30 executions.

C.1.1 Kernels and objective metric

We use the Polybench/GPU benchmark suite kernels to assess the potential for improvement with phase ordering when targeting NVIDIA GPUs. We selected this particular benchmark as it is freely available and thus contributes for making the results presented reproducible. Additionally, the PolyBench/GPU kernels represent well the types of functions commonly executed in general-purpose computing on graphics processing units (GPGPU).

We modified the benchmarks to ensure that the CUDA and OpenCL versions use the same floating-point precision. We performed the minimum of changes to ensure a fair comparison.

Polybench/GPU is a collection of codes implemented for GPUs using CUDA, OpenCL, and HMPP. This benchmark suite includes kernels from 15 benchmarks from different domains which represent computations that would be performed on GPUs in the context of HPC, including convolution kernels (`2DCONV`, `3DCONV`), linear algebra (`2MM`, `3MM`, `ATAX`, `BICG`, `GEMM`, `GESUMMV`, `GRAMSCH`, `MVT`, `SYR2K`, `SYRK`), datamining (`CORR`, `COVAR`), and stencil computations (`FDTD-2D`).

For our experiments we use both the CUDA and the OpenCL implementations available for each PolyBench/GPU benchmark. We rely on the default dataset shape so that reproducibility of our results (e.g., performance improvement using the specialized phase orders presented) is more straightforward.

C.1.2 Compilation and execution flow with specialized phase ordering

We use Clang compiler's OpenCL frontend with the libclc library to generate an LLVM IR representation of a given input OpenCL kernel. The libclc library is an open source library with support for AMDGCN, and R600 and NVPTX targets that implements functions as specified in OpenCL 1.1.

Then, we use the LLVM Optimizer tool (`opt`) to optimize the IR using a specific optimization strategy represented by a compiler phase order, and we link this optimized IR with the libclc OpenCL functions for our target using `llvm-link`. Finally, using Clang, we generate the NVIDIA PTX representation of the kernel from the LLVM bytecode resulting from the previous step, using the `nvptx64-nvidia-nvcl` target. PTX is NVIDIA's intermediate representation for GPU computations, and is used by NVIDIA's OpenCL and CUDA implementations. Although PTX is not the final Assembly executed on the GPU, it is the closest we can get without direct access to the internals of NVIDIA's drivers.

In the code that executes in the host, each PTX representation of the optimized LLVM IR is then used to generate an OpenCL program object instead of loading the OpenCL kernel source file that comes with the specific PolyBench/GPU benchmark. To do this, we used offline compilation.

That is, we load the compiled PTX code with the `clCreateProgramWithBinary` function instead of passing the unmodified OpenCL source code to the `clCreateProgramWithSource` function, which is the most commonly used mechanism in OpenCL.

C.1.3 Validation of the code generated after phase ordering

Each PolyBench/GPU benchmark has verification embedded in its code that consists in executing the OpenCL GPU kernel(s) followed by a functionally equivalent sequential C version on the CPU, and comparing the two. This alone poses a challenge, as CPU executing using the same parameters as the ones used for GPU execution takes a long time for a considerable number of the PolyBench/GPU benchmarks. This would have an unreasonable impact on the phase ordering exploration time.

To reduce the time for each DSE iteration, we separate the validation from the measurement phases. We validate the programs by executing on the CPU and GPU (as in the original Polybench) with inputs that can be processed quickly. However, we also execute the same GPU code using the original inputs in order to measure the execution time.

We further reduce exploration time by checking whether an identical PTX file was previously generated. If so, we reuse the results (i.e., correctness and performance) from that previous execution.

At the end of phase ordering exploration, all compiler pass sequences that were iteratively tested during DSE are ordered by their resulting objective metrics. For the experiments presented here, sequence/metric pairs are ordered from the one resulting in fastest execution time to the one resulting in least performance. Then, as a final validation process, the optimized version that resulted in highest performance is executed with the original inputs on both the non-optimized CPU version and the optimized GPU version, and also with 30 randomly generated inputs that result in the same number of operations. We choose the fastest optimized version that passes validation.

This is performed to eliminate possible situations where a compiled PTX kernel gives correct results using a small input set but gives wrong results with the original input set. This is just a precaution, and in our tests we did not encounter any case where this was a problem.

The PolyBench/GPU kernels are mostly composed of floating-point operations and the result of floating-point operations can be affected by reordering operations and rounding. Because of this we allow for up to 1% difference between the outputs of CPU and GPU executions when testing if a given compiler phase order results in code that generates valid output.

C.2 Performance evaluation

For the experiments presented in this section, the OpenCL kernels from each of the PolyBench/GPU benchmarks were compiled/tested with a set of $10,000$ randomly generated compiler phase orders (the same set was used with all OpenCL codes) composed of 256 LLVM pass instances (can include repeated calls to the same pass). Passes were selected from a list with all LLVM 3.9 passes

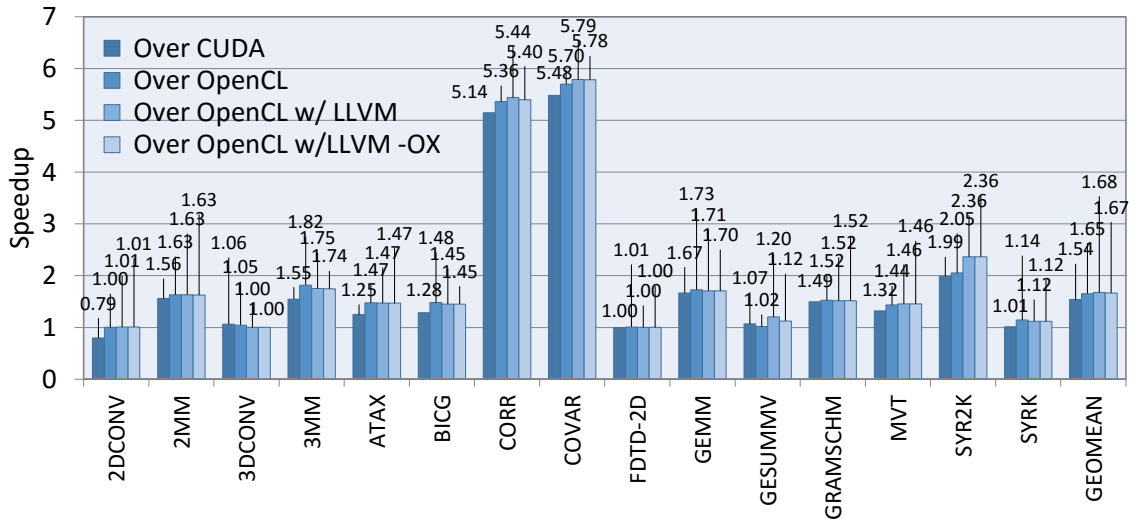

Figure C.1: Performance improvements from phase ordering with LLVM over CUDA implementations and OpenCL using the default compilation pipeline for the NVIDIA GTX1070 GPU and over OpenCL to PTX compilation using Clang/LLVM without (OpenCL w/LLVM) and with standard optimization levels (OpenCL w/LLVM –Ox).

except the ones that resulted in compilation and/or execution problems when used individually to compile the PolyBench/GPU OpenCL kernels.

For each of the benchmarks, we measured the execution times for the CUDA version, the original OpenCL (from source), an offline compiled OpenCL without optimization, an offline compiled OpenCL with standard LLVM optimization levels (i.e., the best of `-O1`, `-O2`, `-O3` and `-Os` for each benchmark, which we will refer to as `-OX`) and an offline compiled OpenCL with our custom compiler optimization phase orders. We compared the results for the various versions of the benchmarks (offline OpenCL versions, OpenCL from source and CUDA) to determine how they perform.

Figure C.1 depicts the performance improvements with phase ordering over the standard OpenCL and CUDA baselines and the other OpenCL baselines.

Using custom phase orders found by iterative compilation produced code that consistently outperforms the other OpenCL variants, and nearly always outperforms the CUDA version. With phase ordering specialization we were able to achieve a geometric mean performance improvement of 1.54× over the CUDA version and a performance improvement of 1.65× over the execution of the OpenCL kernels compiled from source. Additionally, code compiled with specialized phase ordering can be up to 5.48× and up to 5.70× faster than the respective CUDA implementation and the OpenCL compiled from source.

For the tested benchmarks, there were no significant performance difference between the offline compilation model using Clang/LLVM without custom phase ordering and the OpenCL versions from source. Using the LLVM standard optimization level flags did not result in noticeable improvements in terms of the performance of the generated code for most benchmarks. We believe this is because the PTX code is further aggressively optimized by the NVIDIA driver before gener-

ating the final assembly code for the target NVIDIA GPU, so effectively we are using LLVM only as a pre-optimizer. For `2DCONV`, `3DCONV`, `FDTD-2D` and `SYR2K` none of the standard optimization level flags resulted in different code being generated. For benchmarks `2MM`, `3DCONV`, `3MM`, `ATAX`, `BICG`, `GEMM`, `GESUMMV`, `GRAMSCHM`, `MVT` and `SYRK`, the generated code using the optimization level flags differs from the generated code without optimizations. However, the different optimization levels (i.e., `-O1`, `-O2`, `-O3`, `-Os`) all produced the same code.

`CORR` and `COVAR` are the only benchmarks for which different optimization level flags produce different code. However, even in these benchmarks, the performance impact was usually minimal (within 1%). The only exceptions were `GESUMMV` and `GRAMSCHM`. In the case of `GESUMMV`, the use of all tested optimization levels resulted in 1.07× performance improvement over the non-optimized version. For `GRAMSCHM`, the non-optimized version was 1.04× faster than all the versions produced by the optimization level flags.

The difference between the OpenCL baselines is that one represents the defacto OpenCL compilation flow (with compile from source) and the others represent the compilation using LLVM (with compile from binary) using the standard optimization level that results in the generation of code with highest performance on a kernel-by-kernel basis, and compilation using LLVM but with no optimization. Finally, on these benchmarks, performance with CUDA tends to be better than with OpenCL, if no specialized phase ordering is considered. The geometric mean (considering all 15 PolyBench/GPU benchmarks) of the performance improvement with CUDA (over OpenCL from source) is 1.07×. The `2DCONV`, `3MM`, `ATAX`, `BICG` and `SYRK` benchmarks are at least 1.1× faster in CUDA than with OpenCL. All other benchmarks with exception for `3DCONV` and `GESUMMV` are still faster in CUDA than in OpenCL, although by a smaller margin.

Table C.1 depicts LLVM 3.9 compiler phase orders found to have better performance that the OpenCL baseline that relies on Clang/LLVM and the libclc OpenCL library.

C.3 Additional experiments

We performed the following experiments in order to better understand the importance of phase ordering specialization when using LLVM to target NVIDIA GPUs:

1. Test the sequences found for any given PolyBench/GPU OpenCL benchmark (see Table C.1) in the remaining benchmarks.

2. Show how the performance of the different PolyBench/GPU OpenCL changes with different compiler sequences, comparing the performance differences registered for different kernels using the same compiler sequences.

3. Show the effect of the different phase orders, on each benchmark, constructed with permutations of the sequences found for the same benchmark (see Table C.1).

Figure C.2 represents the matrix resulting from testing the sequences from Table C.1 for compilation with each of the PolyBench/GPU OpenCL benchmarks. Values, between 0 and 1 represent

Benchmark	Compiler Phase Order
2MM	`-cfl-anders-aa -dse -loop-reduce -licm -instcombine`
3MM	`-loop-reduce -gvn-hoist -reg2mem -cfl-anders-aa -sroa -licm`
ATAX	`-bb-vectorize -loop-reduce -licm -cfl-anders-aa`
BICG	`-gvn -loop-reduce -cfl-anders-aa -licm -loop-reduce`
CORR	`-cfl-anders-aa -loop-reduce -gvn -sink -loop-extract-single -loop-unswitch -loop-unswitch -ipsccp -reg2mem -licm -nvptx-lower-alloca`
COVAR	`-cfl-anders-aa -loop-unswitch -reassociate -jump-threading -loop-reduce -gvn -loop-unswitch -reassociate -sink -loop-unswitch -loop-reduce -jump-threading -reg2mem -licm -nvptx-lower-alloca`
GEMM	`-cfl-anders-aa -print-memdeps -loop-reduce -licm`
GESUMMV	`-instcombine -reg2mem -mem2reg`
GRAMSCHM	`-sink -reg2mem -licm -cfl-anders-aa -sroa`
MVT	`-gvn -loop-reduce -cfl-anders-aa -licm`
SYR2K	`-loop-reduce -loop-unroll -instcombine -loop-reduce -licm -cfl-anders-aa`
SYRK	`-licm -cfl-anders-aa -reg2mem -licm -sroa`

Table C.1: Compiler phase orders that resulted in compiled kernels with highest performance. Compiler passes that resulted in no performance improvement were eliminated from the compiler phase orders. No compiler phase orders resulted in improving the performance of 2DCONV, 3DCONV or FDTD-2D.

performance factors when compared with the performance obtained using the sequence found to be the best for each benchmark.

The phase orders the exploration process found individually for each benchmark (see Table C.1) result in a very wide performance (from very high to very low) when used to compile the remaining 14 benchmarks. For some benchmark/sequence pairs (`CORR`/`2MM`, `GESUMMV`/`COVAR`, `GESUMMV`/`GRAMSCHM`, `GESUMMV`/`SYRK`) the use of some compiler sequences resulted in incorrect outputs when executing the generated code. These, and any other sequences that result in compiler crashes or execution timeout expiration, are represented on top of the XX axis (i.e., $Y = 0$) in Figure C.2.

Each individual compiler sequence has different impact on different OpenCL kernels. Figure C.3 presents the performance impact (i.e., speedup) of the first 100 individual compiler sequences tested during the initial DSE process (from the set of $10,000$ sequences tested) on each of the OpenCL benchmarks. Each of the points in the charts represents the performance (extracted with 30 executions) for `2DCONV` and for other kernel when compared with baseline. The performance baseline is offline compilation with LLVM with no optimization, as it was experimentally determined that the use of the standard optimization levels in LLVM only very rarely improves the performance of the generated NVIDIA PTX assembly code (see Figure C.1). The performance of the best phase order (see Table C.1) for each OpenCL benchmark is represented as a horizontal

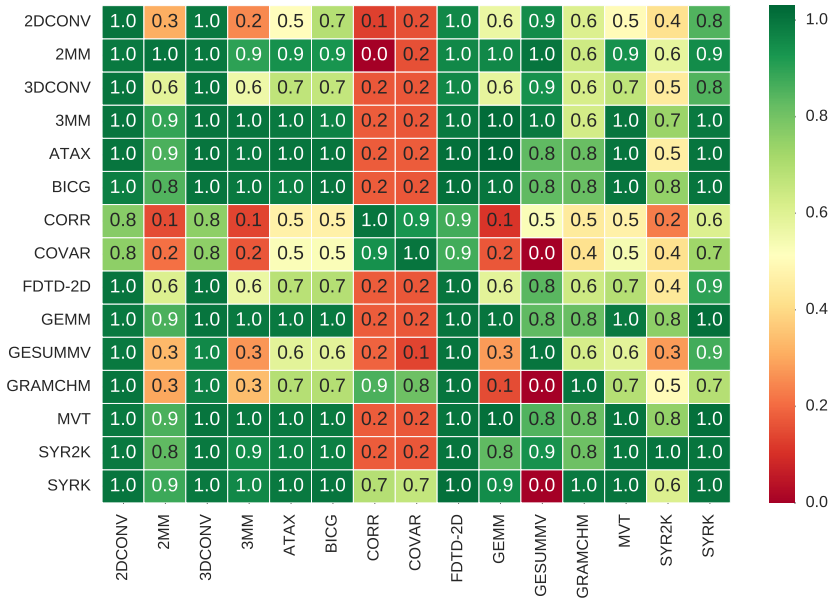

Figure C.2: Performance ratios for using sequences found for each of the benchmark in all benchmarks. The XX axis represents the sequences and the YY axis represents the benchmarks. Performance factors are represented with a precision of 5%. The values represented as 1.0 are in fact between 0.95 and 1.0, inclusive. Values represented as 1.0 but that are closer to 0.95 are represented with a slightly lighter shade of green.

red line. For the `2MM`, `3MM`, `CORR` and `COVAR` OpenCL benchmarks, the first 100 DSE iterations did not allow to achieve the speedup achieved with the $10,000$ iterations. It is also interesting to notice that there is a cluster of points from the compiler sequence solution space close to the baseline (close to horizontal line $y = 0$), and that for some kernels the concentration of points close to the maximum speedup (i.e., close to the red line) is very scarce (e.g., `GEMM`, `GRAMSCHM`, `SYR2K`), while for other kernels this is not the case (e.g., `SYRK`).

Figure C.4 represents the results obtained for experiments where permutations of the phase orders from Table C.1 are tested. Up to $1,000$ randomly generated permutations were tested for each kernel. The only requirement is that each permutations includes all the compiler passes that are present in the phase order from Table C.1 (including the number of pass instances that are repeated in the sequence). The `2DCONV`, `3DCONV` and `FDTD-2D` OpenCL benchmarks are not included because the initial DSE process could not find a phase order that resulted in improving performance. The execution performance after compilation with a large number of permutations resulted in considerable performance degradation. Some of these permutations were only able to achieve 10% or less of the execution performance achieved with the initial phase order (e.g., `3MM`, `CORR`, `COVAR`).

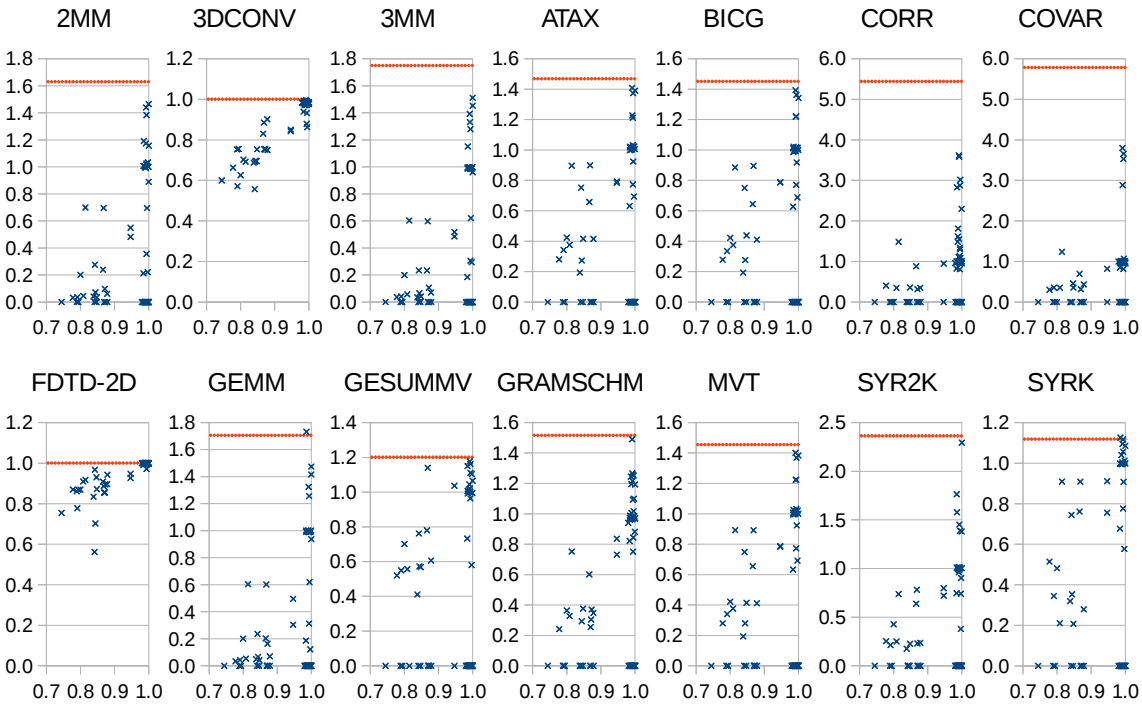

Figure C.3: Speedup for the same sequences on different kernels. All kernels compared with 2DCONV (*XX* axis).

C.4 Explaining performance improvements

We explain for each PolyBench/GPU benchmark what are the reasons behind the performance improvement achieved with phase ordering, comparing with the performance achieved OpenCL baselines and the baseline CUDA versions. More specifically, we compare the PTX output resulting from OpenCL offline compilation with specialized phase ordering with PTX generated from OpenCL offline compilation without phase ordering and with PTX generated from the CUDA versions.

For `2DCONV`, CUDA is 1.26× faster than the OpenCL version optimized with phase ordering. The compiler pass phase order DSE process was not able to find an LLVM sequence capable to optimize this benchmark. The main improvement of CUDA over OpenCL seems to be the generation of more efficient code for loads from global memory. Figure C.5 shows the difference between the two approaches. Whereas load operations typically result in a single CUDA operation, the equivalent for OpenCL typically results in 5 PTX instructions. We believe this difference is the primary reason for CUDA's advantage over OpenCL.

For `2MM`, the OpenCL version optimized with phase ordering is 1.63× and 1.56× faster than the OpenCL and CUDA baselines, respectively. The main reason for this speedup is the removal of store operations within the kernel loop. Both the OpenCL and the CUDA baseline versions of this kernel repeatedly overwrite the same element and this has a negative impact on performance. The phase ordered version instead uses an accumulator register and performs the store only after all the loop computations are complete, which substantially reduces the number of costly mem-

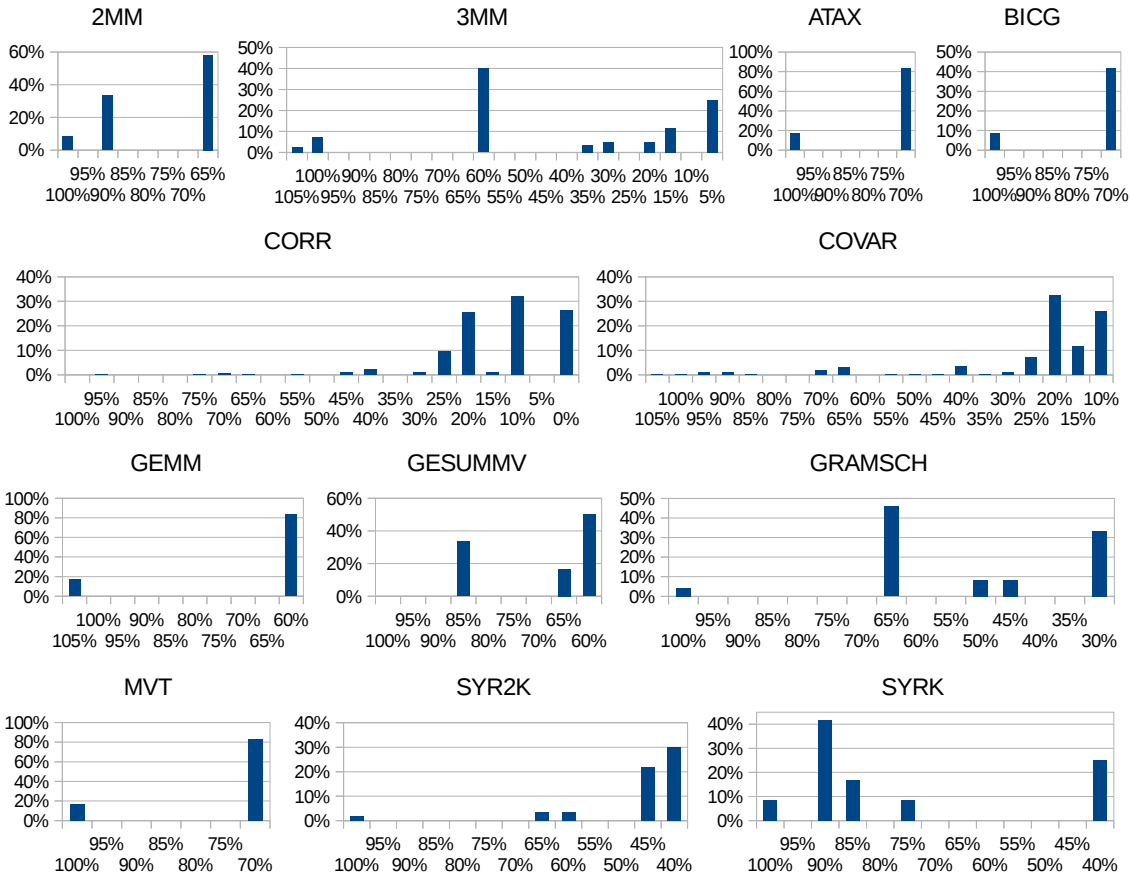

Figure C.4: Percentage of permutations of the best phase order (*YY* axis) that result in given a percentage of performance (*XX* axis) relative to the performance achieved with the best phase order.

```
ld.global.f32   %f2, [%rd6+4]
```

(a) PTX load code generated from CUDA.

```
add.s32   %r17, %r14, %r1;
cvt.s64.s32 %rd16, %r17;
shl.b64   %rd17, %rd16, 2;
add.s64   %rd18, %rd1, %rd17;
ld.global.f32   %f2, [%rd18];
```

(b) PTX load code generated from OpenCL (`-O3`).

Figure C.5: PTX code for equivalent load operations, for CUDA and OpenCL (2DCONV benchmark)

ory accesses. It is unclear why the baseline OpenCL and CUDA versions do not perform this optimization. One possibility is that they are unable to determine that there are no aliasing issues. In the context of this benchmark, this assumption is correct in OpenCL 2.0, as any aliasing would result in a data race, which is undefined behavior (Khronos, 2015). We do not know if the optimization was applied because LLVM correctly discovered this fact, or if there is a bug that

happened to result in correct code by accident. Even if the optimization turns out to be the result of a bug, we believe this speedup represents an opportunity for approaches based on *Loop Versioning* transformations. Although this benchmark uses two kernels, both are equivalent (the only difference being kernel and variable names), and thus the same analysis applies to both. There are two differences between the baseline CUDA and OpenCL compiled versions that can explain the different execution times. The first being the aforementioned issue with load instructions (see Figure C.5), the second being a different loop unroll factor as the phase ordered version based on OpenCL uses efficient load instructions, but also uses a loop unroll factor of 2 (while the CUDA version uses an unrolling factor of 8).

For `3DCONV`, we were unable to achieve a speedup on this benchmark using any of the tested compiler phase orders, when compared with LLVM w/ or w/o the optimization level flags. There is a speedup from the use of the LLVM PTX backend compared with the OpenCL from source compilation path (1.05×) and the compilation from CUDA (1.06×). We believe this happens because most of the time spent on the benchmark is due to global memory loads that are not removed or improved by any LLVM pass. Any optimization will only modify the rest of the code, which takes a negligible amount of time compared to the memory operations.

On the `3MM` benchmark, we were able to achieve speedups of 1.55× and 1.82× over the baseline CUDA and OpenCL version compiled from source, respectively. The main reason for the performance improvement is the removal of the memory store operation from the computation loop.

The OpenCL version of `ATAX` optimized with phase ordering achieves a speedup of 1.47× and 1.25× over the baseline OpenCL and CUDA versions, respectively. Once again, the phase ordered version is able to move memory stores out of the innermost loops of the kernels, which explains the speedups. Additionally, the difference between the CUDA and the baseline OpenCL versions can be explained by a different loop unroll factor (2 for OpenCL, 8 for CUDA). The CUDA version uses the previously described simpler code pattern for memory loads compared to the baseline OpenCL version, but the phase ordering version also uses an efficient memory load pattern.

On the `BICG` benchmark, we were able to achieve a speedup of 1.48× over OpenCL, and 1.28× over CUDA. The main differences between the versions are the memory stores in the kernel loop, the unroll factor and the inefficient memory access patterns in the baseline OpenCL version.

The `CORR` benchmark is one of the benchmarks that benefits the most from phase ordering (5.36× and 5.14× over baseline OpenCL from source and CUDA versions, respectively). The version generated by phase ordering contains several memory accesses to a local memory storage buffer (`__local_depot`) that serves a purpose similar to the stack of the CPU. We believe this is caused by the use of the `reg2mem` pass without a corresponding `mem2reg`. These instructions do not seem to have a significant impact, either because they are eliminated in the compilation of the PTX to device-specific code by the NVIDIA GPU driver, or because accesses to local memory are too fast to affect non-negligible performance variations. Phase ordering is also capable of moving global memory stores out of loops, which neither the CUDA version nor the baseline OpenCL

versions do. In general, for this benchmark, the CUDA version tends to produce more compact load instructions and use higher loop unroll factors than the OpenCL versions.

The `COVAR` and `CORR` benchmarks both use the `mean_kernel` and `reduce_kernel` functions. However, this represents only a fragment of the total execution code, so the compiler sequences for the two benchmarks are different. Regardless, the same conclusions from `CORR` apply to `COVAR`: phase ordering removes global stores from the loop, but introduces several new registers and local memory accesses.

The functions of the `FDTD-2D` benchmark are very straightforward, with little potential for optimization. As such, phase ordering had no impact.

The performance differences for the `GEMM` benchmark (1.67× and 1.73× over the OpenCL from source and the CUDA baselines) can be explained by the removal of the memory store operation from the kernel loop, the different unroll factor and the different pattern of memory load instructions.

There was only a small performance improvement for the `GESUMMV` benchmark (1.07× over CUDA and 1.02× over the baseline OpenCL from source). The phase ordering sequence is able to extract the memory stores out of the main computation loop, but uses a smaller loop unroll factor (2) than the baseline OpenCL and CUDA versions (4 and 16, respectively).

We were able to obtain speedups of 1.49× and 1.52× over the baseline CUDA and OpenCL versions on the `GRAMSCHM`, respectively. Phase ordering is able to move the memory storage operations out of the loop. Aside from that, it uses the same load from memory instruction pattern and unroll factor as the baseline OpenCL version.

The `MVT` benchmark benefits from phase ordering by a factor of 1.32× and 1.44× over the baseline CUDA and OpenCL versions. The main reason for this improvement is the extraction of the store operation from the computation loop.

The `SYR2K` benchmarks benefits from phase ordering by a factor of 1.99× and 2.05× over the baseline CUDA and OpenCL versions, respectively. In general, the same memory load pattern, loop unroll factor and loop invariant memory storage code motion conclusions apply to this benchmark. Phase ordering also seems to outline the segment of the code containing the kernel loop, but this does not seem to be the reason for the performance difference.

For the `SYRK` benchmark, phase ordering improves performance by 1.14× over the OpenCL baselines. We could not achieve significant speedups over the CUDA version. Once again, the main reason for this improvement is the extraction of the store from the loop.

C.5 Problematic phase orders

Figure C.6 depicts the percentage of LLVM compiler phase orders that result in the optimized LLVM IR not being generated, problems with execution of compiled kernels or execution of compiled kernels generation non-expected results.

Considering the phase ordering experiments (`AVERAGE` column) with all PolyBench/GPU benchmarks, the most common problem is the report being non-existent or broken (17%), the

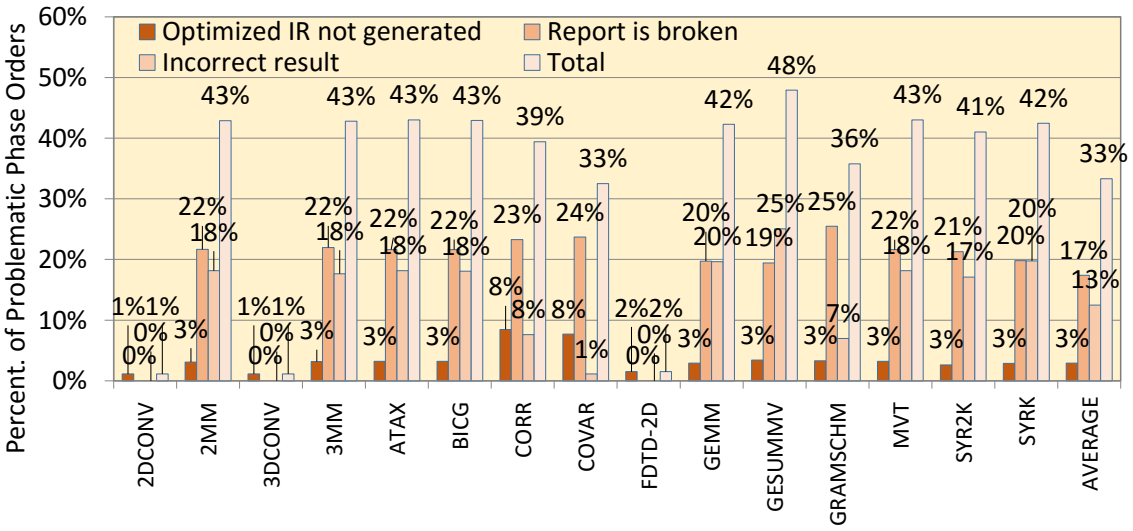

Figure C.6: Percentage of problematic sequences and their kind.

second is the generation of an incorrect result by the compiled OpenCL kernels (13%), and the third (and last) is the optimized LLVM IR not being generated (3%).

Typically, the non-existence of optimized LLVM IR after calling the LLVM Optimizer is related with a compiler crash. In some cases, the execution of the optimized/compiled kernels does not terminate. This can happen because of problems in the kernel themselves or because the compiled kernels are not given enough time to finish execution. Our phase ordering exploration system has a timeout parameter for limiting the overhead of exploration allowed to the execution of the OpenCL kernels compiled after phase ordering. The fact there is a timeout is not detrimental in the sense that it does not result in not finding any given "good" compiler phase order in the context of performance maximization, because if a compiled OpenCL kernel take too much time to execute, it means that it was not compiled with a phase order suitable to maximize performance. Although compiler developers typically make an effort to assure that any given compiler pass that transforms an IR will have as output other IR that is functionally equivalent to the original, this is difficult to implement in practice. Especially, given the fact that compilers can have tens or hundreds of compiler passes and it is difficult to predict all possible interactions between passes. Given the fact that during our phase ordering exploration we are compiling with sequences that were possibly never tested by the compiler developers, there is a greater potential for generating code that does not conform with the same functionality of the original code and that will generate outputs that are different than expected.

C.6 Using features to suggest compiler sequences

It is intuitive to think that the set of compiler passes and compiler sequences that are most suited to compile a given function/program given the target hardware and metrics (e.g., performance, energy, code size) are, at least to some extent, related to the static and/or dynamic features of the

function/program. Taking into account such information when exploring for compiler sequences for a given program/function has the potential to reduce exploration time, or alternatively, achieve better solutions with the same number of compile/test iterations.

We present experiments with a simpler code-feature based approach, where given a function/program we select a number K of sequences from Table C.1 that are assigned to the K functions/programs most similar with it. Those sequences are then used for compilation/evaluation with the new function/program, and the binary resulting in highest performance is selected. This method allows to suggest compiler sequences very efficiently, and with an overhead proportional to the value K selected by the user (i.e., how many compilations/iterations at most does the user tolerate).

C.6.1 Kernel features

The code-features used in the work presented here are extracted with the MILEPOST GCC toolchain (Fursin et al., 2008), which includes the Interactive Compilation Interface (ICI) 2.0 and Machine-Learning (ML) feature extractor version 2.0. The MILEPOST feature extractor that was developed to extract static C code features. We are using the same flow to extract features from the OpenCL PolyBench/C kernels. The host code is not taken into account, as only the OpenCL kernels are being optimized through compiler sequence specialization. Pairs of feature vectors with MILEPOST code features (e.g., feature vectors for function/program from the reference set of functions and feature vector of a given new function/program) are used to compute a similarity metric that determines the compiler sequences that are suggested for evaluation. We did not perform feature selection, thus all 55 code features extracted by MILEPOST-GCC (Fursin et al., 2008) appear in the feature vectors.

C.6.2 Similarity metric

In the experiments presented here we use the cosine distance of the feature vectors associated with the functions/programs as metric of similarity between pairs of OpenCL kernels.

The cosine similarity measures the cosine of the angle between two feature vectors. Given that most features extracted with MILEPOST-GCC represent an absolute count (e.g., number of basic blocks in the method, number of basic blocks with a single successor) or an average of some count (average of number of instructions in basic blocks, average of number of phi-nodes at the beginning of a basic block), it is intuitive to assume that relation between the counts are what is important and not the individual magnitude of the number assigned to each particular feature. The cosine distance is calculated from Equation C.1, where $\vec{a}$ and $\vec{b}$ represent the feature vectors.

$$\cos(\theta) = \frac{\vec{a} \cdot \vec{b}}{\|\vec{a}\|\|\vec{b}\|} \tag{C.1}$$

The cosine similarity can take a value between -1 and 1. It takes a value close to -1 when there is an angle close to 180 degrees between the feature vectors, a close to 0 when the feature

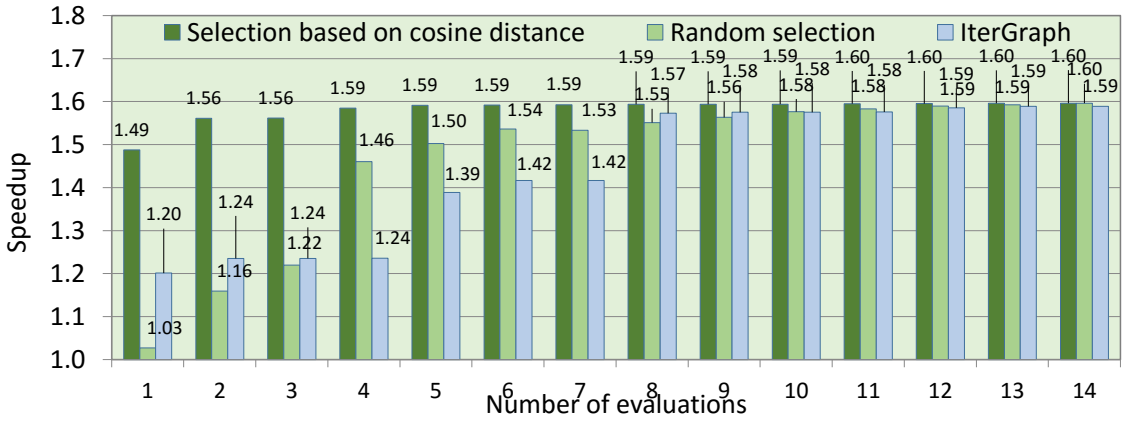

Figure C.7: Performance improvement for using code features in the PolyBench OpenCL kernels to select a number of most similar OpenCL kernels and using their sequences.

vectors make an angle close to 90 degrees, and assumes a value close to 1 when there is a small angle between them.

C.6.3 Experiments using code features

Figure C.7 represents the performance improvement achieved for different numbers of evaluations of compiler sequences from Table C.1. The baseline (i.e., LLVM without any optimization) is used in case no additional sequence evaluated results in better performance. The approach that relies on testing the K (a number given by the programmer/user) sequences associated with the K functions/programs more similar with the new OpenCL kernel is compared with the the random selection of PolyBench/GPU and using the sequences previously found for them. We also compare with the IterGraph approach (see Section 4.2), where the graph was generated from the sequences from Table C.1 using a leave-one-out approach (i.e., when compiling a given OpenCL kernel the sequence associated with that kernel is not used to build the graph). Each random selection is performed $1,000$ times, and the geometric mean of the resulting speedups from using the sequences associated with the selected benchmarks is reported. These experiments are performed with a leave-one-out approach. When a given PolyBench/GPU kernel is the input of exploration, only the other 14 PolyBench/GPU kernels and their sequences are considered.

The results show that the selection of the K most similar OpenCL kernels using the cosine distance (and using the sequence associated with them) results in considerably higher geometric mean performance improvement, for all K, when compared with random selection of K OpenCL kernels (and using their sequences from Table C.1) and also when compared with the IterGraph approach. With only 2 additional sequence evaluations, the method based on the cosine distance results in $1.56\times$ performance improvement, while random selection of OpenCL kernels and using their sequences required 9 evaluations to achieve the same performance improvement. For 14 sequence evaluations, both methods result in the same performance improvement $1.6\times$.

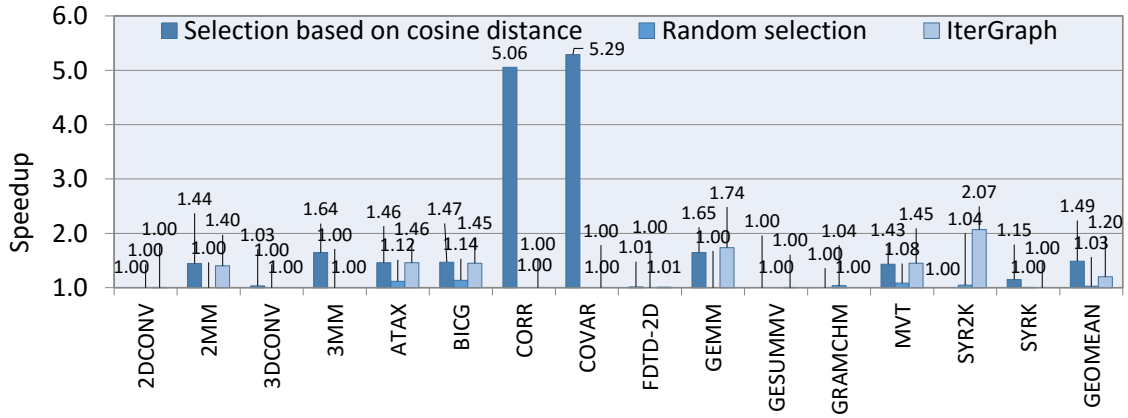

Figure C.8: Selecting the most similar OpenCL kernels and using its sequence.

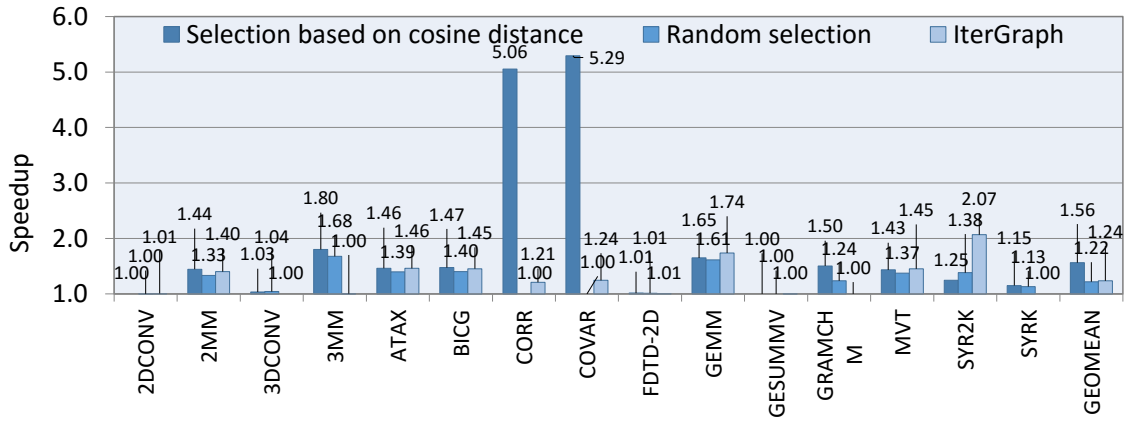

Figure C.9: Selecting the 3 most similar OpenCL kernels and using their sequences.

The IterGraph approach beats using the sequences found for K kernels randomly selected, when considering up do 3 additional sequence evaluations, and after that, only for 8 and 9 additional evaluations. When considering only up to 14 sequence evaluations, the IterGraph approach tends to result in compiled code with reduced performance when compared with the simple code-feature based approach presented here. This may change if considering new OpenCL kernels from a different benchmark or a different domain, as an advantage of the IterGraph approach in comparison with simply evaluating predetermined sequences is that new sequences can be created from the graph (i.e., different from the ones used to generate it).

Figures C.8, C.9 and C.10 present the individual performance improvements when relying only on 1, 3 or 5 additional compiler sequence evaluations.

These results of using the MILEPOST code features with OpenCL C kernels and using cosine similarity as metric for identifying similar OpenCL kernels based on the former are promising. Using only the sequence associated with the most similar OpenCL kernel results in a geomean speedup of 1.49× and 1.03×, for selection based on cosine similarity and for random selection, respectively. For 3 additional sequence evaluations, the speedups improve to 1.56× and 1.22×

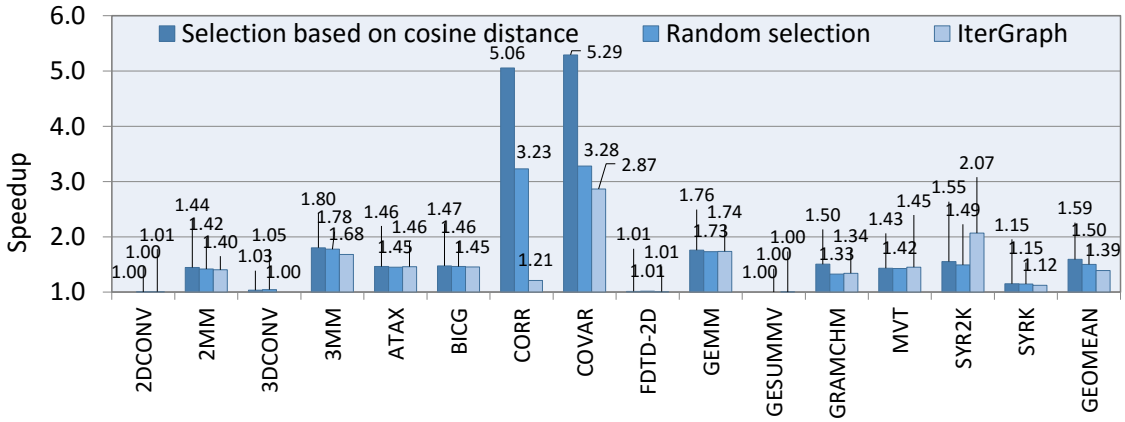

Figure C.10: Selecting the 5 most similar OpenCL kernels and using their sequences.

respectively. Finally, for 5 additional evaluations, the speedups are 1.59× and 1.5× respectively. The speedup using 5 additional evaluations suggested using code features is very close to the highest speedup that is achieved when testing the 14 sequences of all other OpenCL kernels (1.6×).

While the IterGraph approach does not perform very well in comparison with the other two approaches for the small number of evaluations considered in these experiments (i.e., up to 14 evaluations), it could generate, for any number of sequence evaluations (see Figures C.8, C.9 and C.10), compiler optimization sequences that result in higher performance for the `SYR2K` OpenCL kernel than the two other approaches. We believe that when we validate these approaches using OpenCL kernels from a different benchmark (instead of using the leave-one-out approach), there will be more cases where the IterGraph beats the approach for feature-based suggestion of sequences evaluated here.

Appendix D

Phase Ordering for FPGAs

A complete exploration of the design space for software and hardware components, given a set of DSE parameters, can be an infeasible task, due to the large number of alternatives to be explored. This problem is even aggravated when targeting multiple platforms, e.g., different microprocessors and/or reconfigurable hardware such as Field-Programmable Gate Arrays (FPGAs).

Although it is a common practice to apply the same set of optimizations in a fixed order on each method of a program when targeting a given architecture/platform, several researchers have shown the best order of optimizations when targeting FPGAs is function-specific (Huang et al., 2013). Therefore, finding sequences of compiler optimizations that increases performance in FPGAs is a relevant goal.

We present in the following sections, the results of using a Simulated Annealing-based DSE scheme to guide the compilation of two kernels when targeting a MicroBlaze processor and reconfigurable hardware, striving to minimize the latency of the generated design.

D.1 SA-based exploration

For the experiments presented here, given the fact they were performed in an early stage of the PhD, we relied on an older variant of our Simulated Annealing-based DSE scheme to explore multiple compilation design (i.e., software for CPU or VHDL for FPGA) space points generated by distinct compilation sequences. The difference between our early version of the SA-based DSE scheme and the one described in Chapter 4 is that the former does not perform automatic selection of the minimum and maximum temperature (T_{min} and T_{max}), nor the step (α).

For this particular experiments, function $E(s)$ evaluates the energy used by a given configuration, i.e., the latency of a given solution targeting the MicroBlaze processor or reconfigurable hardware, and $temperature(k)$ calculates the next temperature T with a geometric update function (i.e., $T_{n+1} = \alpha T_n$).

The perturbation rules that generate the next optimization sequence to be tested for latency are the following three:

1. Swap two randomly picked optimizations;

2. Replace an optimization by an optimization from the complete optimization list;

3. Replace the unroll factor of a loop by a factor from the list of allowed unroll factors.

Only one perturbation strategy is selected in each iteration of the simulated annealing algorithm. Each perturbation rule has the same probability of being selected, being the last (rule 3) only selected if the loop unrolling engine is used.

D.2 Experiments

The DSE was performed using 34 CoSy engines, including but not limited to constant propagation, loop invariant code motion, common sub-expression elimination, and loop unrolling. Two separated executions of the DSE were performed, without and with the inclusion of the loop unrolling engine. The unroll factor was allowed to arbitrary be set to either one (i.e., no unroll) or two for each loop of the given kernel. Additional loop factors could be explored at the cost of additional exploration time.

Multiple executions of the Simulated Annealing DSE scheme were performed using different values for α, i.e., the temperature decrease multiplication factor. However, here are reported the results obtained for alpha equal to 0.98. The minimum and maximum temperatures (T_{min} and T_{max}) were experimentally chosen to be 0.01 and 100,000, respectively; resulting in the exploration of 798 alternative optimization sequences.

Also, distinct maximum optimization sequence length were tested to determine how it affects performance (i.e., latency) for each kernel, represented by the energy function $E(s)$. A cycle accurate MicroBlaze simulator from ACE[1] and a RTL ModelSim simulator from TUDelft (Nane et al., 2012) were used for simulating the software and hardware implementations, respectively.

Table D.1 depict results for the Filter Subband and Grid Iterate kernels generated when targeting the MicroBlaze processor without loop unrolling.

Maximum length of optimization sequences, resulting speedup and exploration time are respectively represented by $\#$, S and Δt. As expected, the optimization sequences selected by the Simulated Annealing DSE scheme improve the reference latency (i.e., without optimization).

Table D.2 depicts results obtained including the loop unrolling engine when targeting the MicroBlaze soft-core. Unrolling the loops lead to an increase in performance for the both kernels, at the cost of considerably more exploration time (around 65% and 54% for Filter Subband and Grid Iterate respectively). For Grid Iterate, the using loop unrolling results in performance improvements when using up to 8 or 16 optimizations, 7% and 11%, in relation to the reference implementations. For the Filter Subband kernel, the speedup improvements obtained with loop unrolling were 4%, 7% and 9% respectively for sequence composed of up to 4, 8 and 16 engines.

Table D.3 depict results for the Filter Subband and Grid Iterate kernels without loop unrolling when targeting hardware. The DSE took longer when targeting hardware because of longer simulation time.

[1] ACE – Associated Compiler Experts: http://www.ace.nl/

Table D.1: Software optimization sequences found for Filter Subband and Grid Iterate without loop unrolling.

Kernel	#	S	Δt (min)	Optimization Sequence
Filter Subband	4	1.57	17.76	loopinvariant, loopstrength, loopscalar, loopguard
	8	1.68	21.24	loopinvariant, loopscalar, dismemun, loopstrength, strength,looprev, lowerboolval, loopbcoun
	16	1.71	26.39	loopinvariant, cse, copyprop, loopstrength, loopguard, loopscalar,cache, vstrength, dismemun, blockmerge, strength, looprev,loophoist
Grid Iterate	4	1.98	30.15	dismemun, loopinvariant, cse, loopguard
	8	2.07	34.04	noreturn, loopguard, loopinvariant, chainflow, dismemun, cse,cache, strength
	16	2.07	38.53	copyprop, loopguard, cache, loopive, vprop, looprev, dismemun,loopscalar, loopinvariant, demote, promote, cse, strength, lrrename,lowerbitfield

Table D.2: Software optimization sequences found for Filter Subband and Grid Iterate with loop unrolling.

Kernel	#	S	Δt (min)	Optimization Sequence
Filter Subband	4	1.63	29.92	loopinvariant, unroll, loopguard, loopscalar
	8	1.79	32.47	condassigncreate, loopinvariant, loopscalar, dismemun, loopstrength, looprev, unroll, strength
	16	1.86	45.60	loopguard, loopbcount, loopstrength, mvpostop, setrefobj, dismemun, loopinvariant, loopfuse, lrrename, loopscalar, promote, vprop, funceval, strength, looprev, unroll
Grid Iterate	4	1.74	47.50	loopinvariant, alias, unroll, loopguard
	8	2.21	50.03	dismemun, unroll, constprop, conevun, loopinvariant, cse, lowerpfc, loopguard
	16	2.29	60.64	ifconvert, dismemun, exprprop, constprop, unroll, unroll, algebraic, loopinvariant, loopguard, markconvert, demote, tailmerge, cse, tailrec, strength, conevun

Table D.3: Hardware optimization sequences found for Filter Subband and Grid Iterate without loop unrolling

Kernel	#	S	Δt (min)	Optimization Sequence
Filter Subband	4	1.13	168.60	loopscalar, cache, loopremove, loopstrength
	8	1.13	171.34	scalarreplace, loopscalar, noreturn, loopstrength, cache, exprprop, hwloopcreate, conevun
	16	1.13	175.18	tailrec, loopscalar, alias, conevun, setpurity, cache, loophoist, loopstrength, lowerpfc, funceval, lowerboolval, hwloopcreate
Grid Iterate	4	1.11	216.31	loopstrength, loopbcount, promote, loopinvariant
	8	1.11	221.98	lowerboolval, setpurity, cse, lowerbitfield, loopstrength, vstrength, loopinvariant, loopbcount
	16	1.11	224.32	loopive, cse, globcse, loopstrength, loopinvariant, misc, conevun, loopbcount, noreturn, promote, ckfstrength, loopcanon, dismemun

Table D.4 depicts the results using loop unrolling when targeting hardware for the Filter Subband kernel. Speedups over reference improve for all up to k sequence lengths in relation to experiments without loop unrolling, 5% for up to 4 and 11% for up to 8 and 16 optimizations, at the cost of an average exploration time increase around 6%.

More significant speedups were achieved in software than in hardware and generally using more engines resulted in performance improvements. Additional experiments considering the exploration of 395 (alpha equal to 0:96) instead of 798 (alpha equal to 0:98) alternative optimization sequences, resulted in reducing the exploration time to half while the performance degradation was only around 7% and 5% when targeting filter subband and Grid Iterate for software with loop unrolling; with no performance degradation when targeting hardware.

Table D.4: Hardware optimization sequences found by SA for Filter Subband with loop unrolling.

Kernel	#	S	Δt (min)	Optimization Sequence
Filter Subband	4	1.19	176.95	loopinvariant, unroll, cache, loopguard
	8	1.25	171.98	exprprop, loopstrength, loopguard, conevun, unroll, loopscalar, cache, promote
	16	1.25	197.76	ckfstrength, condassigncreate, loopguard, loopinvariant, loopstrength, tailrec, loopfuse, cse, unroll, loopstrength, lrrename, cache, loopcanon, loopremove, loopbcount, scalarreplace

References

ACE. CoSy compiler development system. http://www.ace.nl/compiler/cosy.html [online].

ACE (2008). CoSy CCMIR definition. Technical Report CoSy-8002-CCMIR.

Agakov, F., Bonilla, E., Cavazos, J., Franke, B., Fursin, G., O'Boyle, M. F. P., Thomson, J., Toussaint, M., and Williams, C. K. I. (2006). Using machine learning to focus iterative optimization. In *International Symposium on Code Generation and Optimization (CGO'06)*, pages 11 pp.–.

Almagor, L., Cooper, K. D., Grosul, A., Harvey, T. J., Reeves, S. W., Subramanian, D., Torczon, L., and Waterman, T. (2004). Finding effective compilation sequences. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES '04, pages 231–239, New York, NY, USA. ACM.

Alpern, B., Attanasio, C. R., Barton, J. J., Burke, M. G., Cheng, P., Choi, J.-D., Cocchi, A., Fink, S. J., Grove, D., Hind, M., Hummel, S. F., Lieber, D., Litvinov, V., Mergen, M. F., Ngo, T., Russell, J. R., Sarkar, V., Serrano, M. J., Shepherd, J. C., Smith, S. E., Sreedhar, V. C., Srinivasan, H., and Whaley, J. (2000). The jalapeño virtual machine. *IBM Syst. J.*, 39(1):211–238.

Altera Software (2016). Modelsim. https://www.altera.com/products/design-software/model---simulation/modelsim-altera-software.html [online].

ARM. Cortex-M4 processor. http://www.arm.com/products/processors/cortex-m/cortex-m4-processor.php [online].

ARM Limited (2013). big.LITTLE technology: The future of mobile. https://www.arm.com/files/pdf/big_LITTLE_Technology_the_Futue_of_Mobile.pdf [online].

Ashouri, A. H., Mariani, G., Palermo, G., and Silvano, C. (2014). A bayesian network approach for compiler auto-tuning for embedded processors. In *2014 IEEE 12th Symposium on Embedded Systems for Real-time Multimedia (ESTIMedia)*, pages 90–97.

Bae, H., Mustafa, D., Lee, J.-W., Aurangzeb, Lin, H., Dave, C., Eigenmann, R., and Midkiff, S. P. (2013). The Cetus source-to-source compiler infrastructure: Overview and evaluation. *International Journal of Parallel Programming*, 41(6):753–767.

Bau, D. (2016). seedrandom.js. https://github.com/davidbau/seedrandom [online].

Baxter, I. D., Pidgeon, C., and Mehlich, M. (2004). DMS®: Program transformations for practical scalable software evolution. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 625–634, Washington, DC, USA. IEEE Computer Society.

Baxter, J. and Kristiansson, S. mor1kx - an OpenRISC 1000 processor IP core. https://github.com/openrisc/mor1kx [online].

BCC Research (2016). Embedded systems: Technologies and markets. https://www.bccresearch.com/market-research/information-technology/embedded-systems-techs-markets-report-ift016f.html [online].

Benitez, M. E. and Davidson, J. W. (1988). A portable global optimizer and linker. *SIGPLAN Notices - Proceedings of the SIGPLAN '88 conference on Programming language design and implementation*, 23(7):329–338.

Betkaoui, B., Thomas, D. B., and Luk, W. (2010). Comparing performance and energy efficiency of FPGAs and GPUs for high productivity computing. In *2010 International Conference on Field-Programmable Technology*, pages 94–101.

Bispo, J., Pinto, P., Nobre, R., Carvalho, T., Cardoso, J. M. P., and Diniz, P. C. (2013). The MATISSE MATLAB compiler. In *2013 11th IEEE International Conference on Industrial Informatics (INDIN)*, pages 602–608.

Blackburn, S. M., Garner, R., Hoffmann, C., Khang, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J. E. B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., and Wiedermann, B. (2006). The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 169–190, New York, NY, USA. ACM.

Bodin, F., Kisuki, T., Knijnenburg, P., O' Boyle, M., and Rohou, E. (1998). Iterative compilation in a non-linear optimisation space. In *Workshop on Profile and Feedback-Directed Compilation*, Paris, France.

Bondhugula, U., Hartono, A., Ramanujam, J., and Sadayappan, P. (2008). A practical automatic polyhedral parallelizer and locality optimizer. *SIGPLAN Notices - Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 43(6):101–113.

Bravenboer, M., Kalleberg, K. T., Vermaas, R., and Visser, E. (2008). Stratego/XT 0.17. a language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70.

Canis, A., Choi, J., Aldham, M., Zhang, V., Kammoona, A., Anderson, J. H., Brown, S., and Czajkowski, T. (2011). LegUp: High-level synthesis for FPGA-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, pages 33–36, New York, NY, USA. ACM.

Canis, A., Choi, J., Aldham, M., Zhang, V., Kammoona, A., Czajkowski, T., Brown, S. D., and Anderson, J. H. (2013). LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems. *Transactions on Embedded Computing Systems (TECS) - Special issue on application-specific processors*, 13(2):24:1–24:27.

Cardoso, J. M. P., Carvalho, T., Coutinho, J. G. F., Luk, W., Nobre, R., Diniz, P. C., and Petrov, Z. (2012a). LARA: An aspect-oriented programming language for embedded systems. In *Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development*, AOSD '12, pages 179–190, New York, NY, USA. ACM.

Cardoso, J. M. P., Carvalho, T., Coutinho, J. G. F., Nobre, R., Nane, R., Diniz, P. C., Petrov, Z., Luk, W., and Bertels, K. (2013a). Controlling a complete hardware synthesis toolchain with LARA aspects. *Microprocessors and Microsystems*, 37(8, Part C):1073 – 1089. Special Issue on European Projects in Embedded System Design: EPESD2012.

Cardoso, J. M. P., Coutinho, J. G. F., Nane, R., Sima, V.-M., Olivier, B., Carvalho, T., Nobre, R., Diniz, P. C., Petrov, Z., Bertels, K., Gonçalves, F., van Someren, H., Hübner, M., Constantinides, G., Luk, W., Becker, J., Krátký, K., Bhattacharya, S., Alves, J. C., and Ferreira, J. C. (2013b). *The REFLECT Design-Flow*, pages 13–34. Springer New York, New York, NY.

Cardoso, J. M. P., Fernandes, J. M., Monteiro, M. P., Carvalho, T., and Nobre, R. (2013c). Enriching MATLAB with aspect-oriented features for developing embedded systems. *Journal of Systems Architecture*, 59(7):412 – 428.

Cardoso, J. M. P., Teixeira, J., Alves, J. C., Nobre, R., Diniz, P. C., Coutinho, J. G. F., and Luk, W. (2012b). Specifying compiler strategies for FPGA-based systems. In *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, pages 192–199.

Cavazos, J. and O'Boyle, M. F. P. (2006). Method-specific dynamic compilation using logistic regression. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 229–240, New York, NY, USA. ACM.

Chabbi, M. M., Mellor-Crummey, J. M., and Cooper, K. D. (2011). Efficiently exploring compiler optimization sequences with pairwise pruning. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EXADAPT '11, pages 34–45, New York, NY, USA. ACM.

Chen, Y., Fang, S., Huang, Y., Eeckhout, L., Fursin, G., Temam, O., and Wu, C. (2012). Deconstructing iterative optimization. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(3):21:1–21:30.

Christen, M., Schenk, O., and Burkhart, H. (2011). PATUS: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS'11, pages 676–687, Washington, DC, USA. IEEE Computer Society.

Cobham Gaisler AB. LEON3 processor. http://www.gaisler.com/index.php/products/processors/leon3 [online].

Cobham Gaisler AB. TSIM2 - ERC32/LEON simulator. http://www.gaisler.com/index.php/products/simulators/tsim [online].

Cobham Gaisler AB (2016). TSIM2 simulator user's manual. Technical Report TSIM2-UM (ver. 2.0.46).

Coleman, S. and McKinley, K. S. (1995). Tile size selection using cache organization and data layout. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, PLDI '95, pages 279–290, New York, NY, USA. ACM.

Cooper, K. D., Grosul, A., Harvey, T. J., Reeves, S., Subramanian, D., Torczon, L., and Waterman, T. (2006). Exploring the structure of the space of compilation sequences using randomized search algorithms. *The Journal of Supercomputing*, 36(2):135–151.

Cooper, K. D., Schielke, P. J., and Subramanian, D. (1999). Optimizing for reduced code space using genetic algorithms. In *Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems*, LCTES '99, pages 1–9, New York, NY, USA. ACM.

Cooper, K. D., Subramanian, D., and Torczon, L. (2002). Adaptive optimizing compilers for the 21st century. *The Journal of Supercomputing*, 23(1):7–22.

Cordy, J. R. (2004). TXL - a language for programming language tools and applications. *Electronic Notes in Theoretical Computer Science*, 110:3 – 31.

Coutinho, J. G. F., Cardoso, J. M. P., Carvalho, T., Nobre, R., Bhattacharya, S., Diniz, P. C., Fitzpatrick, L., and Nane, R. (2013). *Deriving Resource Efficient Designs Using the REFLECT Aspect-Oriented Approach*, pages 226–228. Springer Berlin Heidelberg, Berlin, Heidelberg.

Coutinho, J. G. F., Carvalho, T., Durand, S., and Cardoso, J. M. P. (2012a). The LARA Aspect-IR. REFLECT internal technical report. Technical report.

Coutinho, J. G. F., Carvalho, T., Durand, S., Cardoso, J. M. P., Nobre, R., Diniz, P. C., and Luk, W. (2012b). Experiments with the LARA aspect-oriented approach. In *Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development Companion*, AOSD Companion '12, pages 27–30, New York, NY, USA. ACM.

Darte, A. (1999). On the complexity of loop fusion. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, PACT '99, pages 149–, Washington, DC, USA. IEEE Computer Society.

Dave, C., Bae, H., Min, S. J., Lee, S., Eigenmann, R., and Midkiff, S. (2009). Cetus: A source-to-source compiler infrastructure for multicores. *Computer*, 42(12):36–42.

David, H., Gorbatov, E., Hanebutte, U. R., Khanna, R., and Le, C. (2010). RAPL: Memory power estimation and capping. In *Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED '10, pages 189–194, New York, NY, USA. ACM.

Diniz, P. C. and Rinard, M. C. (1997). Dynamic feedback: An effective technique for adaptive computing. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, PLDI '97, pages 71–84, New York, NY, USA. ACM.

Donadio, S., Brodman, J., Roeder, T., Yotov, K., Barthou, D., Cohen, A., Garzarán, M. J., Padua, D., and Pingali, K. (2006). A language for the compact representation of multiple program versions. In *Proceedings of the 18th International Conference on Languages and Compilers for Parallel Computing*, LCPC'05, pages 136–151, Berlin, Heidelberg. Springer-Verlag.

EEMBC. About CoreMark. http://www.eembc.org/coremark/about.php [online].

Eide, E. and Regehr, J. (2008). *Volatiles are miscompiled, and what to do about it*, pages 255–264.

Forsythe, G. E., Malcolm, M. A., and Moler, C. B. (1977). *Computer Methods for Mathematical Computations*. Prentice Hall Professional Technical Reference.

Free Software Foundation. Auto-vectorization in GCC. https://gcc.gnu.org/projects/tree-ssa/vectorization.html [online].

Free Software Foundation. GCC command options - options that control optimization. https://gcc.gnu.org/onlinedocs/gcc-4.9.1/gcc/Optimize-Options.html [online].

Free Software Foundation. GNU compiler collection. https://www.gnu.org/software/gcc/ [online].

Frigo, M. and Johnson, S. G. (2005). The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231.

Fursin, G., Kashnikov, Y., Memon, A. W., Chamski, Z., Temam, O., Namolaru, M., Yom-Tov, E., Mendelson, B., Zaks, A., Courtois, E., Bodin, F., Barnard, P., Ashton, E., Bonilla, E., Thomson, J., Williams, C. K. I., and O'Boyle, M. (2011). Milepost GCC: Machine learning enabled self-tuning compiler. *International Journal of Parallel Programming*, 39(3):296–327.

Fursin, G., Miranda, C., Temam, O., Namolaru, M., Yom-Tov, E., Zaks, A., Mendelson, B., Bonilla, E., Thomson, J., Leather, H., Williams, C., O'Boyle, M., Barnard, P., Ashton, E., Courtois, E., and Bodin, F. (2008). MILEPOST GCC: machine learning based research compiler. In *GCC Summit*, Ottawa, Canada.

Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition.

Gonçalves, F., Petrov, Z., de F. Coutinho, J. G., Nane, R., Sima, V.-M., Cardoso, J. M. P., Werner, S., Bhattacharya, S., Carvalho, T., Nobre, R., de Sá, J., Teixeira, J., Diniz, P. C., Bertels, K., Constantinides, G., Luk, W., Becker, J., Alves, J. C., Ferreira, J. C., and Almeida, G. M. (2013). *LARA Experiments*, pages 135–179. Springer New York, New York, NY.

Gradecki, J. D. and Lesiecki, N. (2003). *Mastering AspectJ: Aspect-Oriented Programming in Java*. John Wiley & Sons, Inc., New York, NY, USA.

Gries, M. (2004). Methods for evaluating and covering the design space during early design development. *Integration, the VLSI Journal*, 38(2):131 – 183.

Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., and Brown, R. B. (2001). MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, WWC '01, pages 3–14, Washington, DC, USA. IEEE Computer Society.

Hackenberg, D., Schöne, R., Ilsche, T., Molka, D., Schuchart, J., and Geyer, R. (2015). An energy efficiency feature survey of the Intel Haswell processor. In *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, IPDPSW'15, pages 896–904, Washington, DC, USA. IEEE Computer Society.

Hall, M., Chame, J., Chen, C., Shin, J., Rudy, G., and Khan, M. M. (2010). *Loop Transformation Recipes for Code Generation and Auto-Tuning*, pages 50–64. Springer Berlin Heidelberg, Berlin, Heidelberg.

Hansen, E. and Walster, G. (2003). *Global Optimization Using Interval Analysis: Revised And Expanded*. Monographs and textbooks in pure and applied mathematics. CRC Press.

Hara, Y., Tomiyama, H., Honda, S., Takada, H., and Ishii, K. (2008). CHStone: A benchmark program suite for practical c-based high-level synthesis. In *2008 IEEE International Symposium on Circuits and Systems*, pages 1192–1195.

Hardkernel. ODROID XU+E. http://www.hardkernel.com/main/products/prdt_info.php?g_code=G137463363079 [online].

Harris, M. (2008). Many-core GPU computing with NVIDIA CUDA. In *Proceedings of the 22Nd Annual International Conference on Supercomputing*, ICS '08, pages 1–1, New York, NY, USA. ACM.

Hartono, A., Norris, B., and Sadayappan, P. (2009). Annotation-based empirical performance tuning using Orio. In *2009 IEEE International Symposium on Parallel Distributed Processing*, pages 1–11.

Hendrix, E. M. T. and G.-Tóth, B. (2010). *Introduction to Nonlinear and Global Optimization*. Springer New York, New York, NY.

Horst, R. (2002). *Introduction to Global Optimization (Nonconvex Optimization and Its Applications)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Hoste, K. and Eeckhout, L. (2007). Microarchitecture-independent workload characterization. *IEEE Micro*, 27(3):63–72.

Huang, Q., Lian, R., Canis, A., Choi, J., Xi, R., Brown, S., and Anderson, J. (2013). The effect of compiler optimizations on high-level synthesis for FPGAs. In *2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 89–96.

Huang, Q., Lian, R., Canis, A., Choi, J., Xi, R., Calagar, N., Brown, S., and Anderson, J. (2015). The effect of compiler optimizations on high-level synthesis-generated hardware. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 8(3):14:1–14:26.

Intel Corporation. Intel Turbo Boost technology 2.0. http://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html [online].

Jantz, M. R. and Kulkarni, P. A. (2010). Eliminating false phase interactions to reduce optimization phase order search space. In *Proceedings of the 2010 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '10, pages 187–196, New York, NY, USA. ACM.

Jantz, M. R. and Kulkarni, P. A. (2013a). Exploiting phase inter-dependencies for faster iterative compiler optimization phase order searches. In *Proceedings of the 2013 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '13, pages 7:1–7:10, Piscataway, NJ, USA. IEEE Press.

Jantz, M. R. and Kulkarni, P. A. (2013b). Performance potential of optimization phase selection during dynamic JIT compilation. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '13, pages 131–142, New York, NY, USA. ACM.

Jiménez, M., Llabería, J. M., and Fernández, A. (2002). Register tiling in nonrectangular iteration spaces. *ACM Trans. Program. Lang. Syst.*, 24(4):409–453.

Khronos (2015). The OpenCL C specification, version 2.0. Technical report.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). *Aspect-oriented programming*, pages 220–242. Springer Berlin Heidelberg, Berlin, Heidelberg.

Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P. (1983). Optimization by simulated annealing. *SCIENCE*, 220(4598):671–680.

Klemm, M. (2008). The thing from another world (or: How do OpenMP compilers work? part 1). http://www.drdobbs.com/parallel/how-do-openmp-compilers-work-part-1/226300148 [online].

Kulkarni, P., Hines, S., Hiser, J., Whalley, D., Davidson, J., and Jones, D. (2004). Fast searches for effective optimization phase sequences. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 171–182, New York, NY, USA. ACM.

Kulkarni, P. A., Jantz, M. R., and Whalley, D. B. (2010). Improving both the performance benefits and speed of optimization phase sequence searches. In *Proceedings of the ACM SIGPLAN/SIGBED 2010 Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES '10, pages 95–104, New York, NY, USA. ACM.

Kulkarni, P. A., Whalley, D. B., Tyson, G. S., and Davidson, J. W. (2009). Practical exhaustive optimization phase order exploration and evaluation. *ACM Transactions on Architecture and Code Optimization (TACO)*, 6(1):1:1–1:36.

Kulkarni, S. and Cavazos, J. (2012). Mitigating the compiler optimization phase-ordering problem using machine learning. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 147–162, New York, NY, USA. ACM.

Lee, C., Potkonjak, M., and Mangione-Smith, W. H. (1997). MediaBench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 30, pages 330–335, Washington, DC, USA. IEEE Computer Society.

Lee, S.-I., Johnson, T. A., and Eigenmann, R. (2004). *Cetus – An Extensible Compiler Infrastructure for Source-to-Source Transformation*, pages 539–553. Springer Berlin Heidelberg, Berlin, Heidelberg.

Leroy, X. (2009). Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115.

Leroy, X. (2011). Verifying a compiler: Why? how? how far? http://www.cgo.org/cgo2011/Xavier_Leroy.pdf [online].

Leupers, R. (1997). *Retargetable Code Generation for Digital Signal Processors*. Kluwer Academic Publishers, Norwell, MA, USA.

Li, Y., Zhang, Y.-Q., Liu, Y.-Q., Long, G.-P., and Jia, H.-P. (2013). MPFFT: An auto-tuning FFT library for OpenCL GPUs. *Journal of Computer Science and Technology*, 28(1):90–105.

Liu, Q., Todman, T., de F. Coutinho, J. G., Luk, W., and Constantinides, G. A. (2009). Optimising designs by combining model-based and pattern-based transformations. In *2009 International Conference on Field Programmable Logic and Applications*, pages 308–313.

LLVM Developer Group. clang: a C language family frontend for LLVM. http://clang.llvm.org/ [online].

LLVM Developer Group. The LLVM compiler infrastructure. http://llvm.org/ [online].

LLVM Developer Group. llvm-gcc - LLVM C front-end. http://releases.llvm.org/2.8/docs/CommandGuide/html/llvmgcc.html [online].

LLVM Developer Group. OpenMP: Support for the OpenMP language. http://openmp.llvm.org/ [online].

LLVM Developer Group. opt - LLVM optimizer. http://llvm.org/docs/CommandGuide/opt.html [online].

LLVM Developer Group. Performance tips for frontend authors. http://releases.llvm.org/3.9.0/docs/Frontend/PerformanceTips.html [online].

Luk, W., Coutinho, J. G. F., Todman, T., Lam, Y. M., Osborne, W., Susanto, K. W., Liu, Q., and Wong, W. S. (2009). A high-level compilation toolchain for heterogeneous systems. In *2009 IEEE International SOC Conference (SOCC)*, pages 9–18.

Magni, A., Grewe, D., and Johnson, N. (2013). Input-aware auto-tuning for directive-based GPU programming. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, GPGPU-6, pages 66–75, New York, NY, USA. ACM.

Martins, L. G. A., Nobre, R., Cardoso, J. M. P., Delbem, A. C. B., and Marques, E. (2016). Clustering-based selection for the exploration of compiler optimization sequences. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13(1):8:1–8:28.

Martins, L. G. A., Nobre, R., Delbem, A. C. B., Marques, E., and Cardoso, J. M. P. (2014a). A clustering-based approach for exploring sequences of compiler optimizations. In *2014 IEEE Congress on Evolutionary Computation (CEC)*, pages 2436–2443.

Martins, L. G. A., Nobre, R., Delbem, A. C. B., Marques, E., and Cardoso, J. M. P. (2014b). Exploration of compiler optimization sequences using clustering-based selection. In *Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, LCTES '14, pages 63–72, New York, NY, USA. ACM.

Mccarthy, J. and Painter, J. (1967). Correctness of a compiler for arithmetic expressions. pages 33–41. American Mathematical Society.

McGuire, M. (2014). CS371: Computational graphics. Williams College, Department of Computer Science. Williamstown, MA, USA.

McMahon, F. H. (1986). Livermore Fortran kernels: A computer test of numerical performance range. Technical Report UCRL-53745, Lawrence Livermore National Laboratory, Livermore, CA.

Mirkovic, D. (2001). Automatic performance tuning in the UHFFT library. In *Proceedings of the International Conference on Computational Sciences-Part I*, ICCS '01, pages 71–80, London, UK, UK. Springer-Verlag.

Nane, R., Sima, V. M., Olivier, B., Meeuws, R., Yankova, Y., and Bertels, K. (2012). DWARV 2.0: A CoSy-based C-to-VHDL hardware compiler. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*, pages 619–622.

Nickolls, J., Buck, I., Garland, M., and Skadron, K. (2008). Scalable parallel programming with CUDA. *Queue*, 6(2):40–53.

Nobre, R. (2013). Identifying sequences of optimizations for HW/SW compilation. In *2013 23rd International Conference on Field programmable Logic and Applications*, pages 1–2.

Nobre, R., Cardoso, J. M. P., and Alves, J. C. (2014a). A DSE example of using LARA for identifying compiler optimization sequences. In *X Jornadas sobre Sistemas Reconfiguráveis*.

Nobre, R., Cardoso, J. M. P., Olivier, B., Nane, R., Fitzpatrick, L., Coutinho, J. G. F., van Someren, H., Sima, V.-M., Bertels, K., and Diniz, P. C. (2013a). *Hardware/Software Compilation*, pages 105–134. Springer New York, New York, NY.

Nobre, R., Martins, L. G. A., and Cardoso, J. M. P. (2015). Use of previously acquired positioning of optimizations for phase ordering exploration. In *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems*, SCOPES '15, pages 58–67, New York, NY, USA. ACM.

Nobre, R., Martins, L. G. A., and Cardoso, J. M. P. (2016a). A graph-based iterative compiler pass selection and phase ordering approach. In *Proceedings of the 17th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, Tools, and Theory for Embedded Systems*, LCTES 2016, pages 21–30, New York, NY, USA. ACM.

Nobre, R., Pinto, P., Carvalho, T., Cardoso, J. M. P., and Diniz, P. C. (2013b). LARA-based strategies for targeting multicore architectures. In *Proceedings of 17th Workshop on Compilers for Parallel Computers*, CPC '13.

Nobre, R., Pinto, P., Carvalho, T., Cardoso, J. M. P., and Diniz, P. C. (2014b). On expressing strategies for directive-driven multicore programing models. In *Proceedings of Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms*, PARMA-DITAM '14, pages 7:7–7:12, New York, NY, USA. ACM.

Nobre, R., Reis, L., and Cardoso, J. M. (2016b). Compiler phase ordering as an orthogonal approach for reducing energy consumption. In *Proceedings of the 19th Workshop on Compilers for Parallel Computing*, CPC '16.

Nobre, R., Reis, L., and Cardoso, J. M. P. (2017). Impact of compiler phase ordering when targeting GPUs. In *Proceedings of the 15th International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms*, HeteroPar'2017.

NVIDIA Corporation (2015). Nvidia G-SYNC - technology.

Olukotun, K. and Hammond, L. (2005). The future of microprocessors. *Queue*, 3(7):26–29.

Oracle (2016). Berkeley DB. http://www.oracle.com/technology/products/berkeley-db/index.html [online].

Padua, D. A. and Wolfe, M. J. (1986). Advanced compiler optimizations for supercomputers. *Commun. ACM*, 29(12):1184–1201.

Palermo, G., Silvano, C., and Zaccaria, V. (2005). Multi-objective design space exploration of embedded systems. *Journal of Embedded Computing - Low-power Embedded Systems*, 1(3):305–316.

Palermo, G., Silvano, C., and Zaccaria, V. (2009). ReSPIR: A response surface-based pareto iterative refinement for application-specific design space exploration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(12):1816–1829.

Pinto, P., Abreu, R., and Cardoso, J. M. P. (2015). Fault detection in C programs using monitoring of range values: Preliminary results. *CoRR*, abs/1505.01878.

Pouchet, L.-N. and Bondugula, U. PolyBench 3.0 - the polyhedral benchmark suite. http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/ [online].

Pouchet, L.-N., Bondugula, U., and Yuki, T. PolyBench/C 4.1 - the polyhedral benchmark suite. http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/ [online].

Purini, S. and Jain, L. (2013). Finding good optimization sequences covering program space. *ACM Transactions on Architecture and Code Optimization (TACO) - Special Issue on High-Performance Embedded Architectures and Compilers*, 9(4):56:1–56:23.

Puschel, M., Moura, J. M. F., Johnson, J. R., Padua, D., Veloso, M. M., Singer, B. W., Xiong, J., Franchetti, F., Gacic, A., Voronenko, Y., Chen, K., Johnson, R. W., and Rizzolo, N. (2005). SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE*, 93(2):232–275.

Qasem, A., Jin, G., and Mellor-crummey, J. (2003). Improving performance with integrated program transformations. Technical report, In manuscript, Rice University, Houston, Texas.

Qian, Y., Carr, S., and Sweany, P. (2002). Loop fusion for clustered VLIW architectures. In *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems: Software and Compilers for Embedded Systems*, LCTES/SCOPES '02, pages 112–119, New York, NY, USA. ACM.

REFLECT. FP7 project. http://www.reflect-project.eu/ [offline].

Renganarayanan, L., Kim, D., Strout, M. M., and Rajopadhye, S. (2012). Parameterized loop tiling. *ACM Trans. Program. Lang. Syst.*, 34(1):3:1–3:41.

Reyes, R., López, I., Fumero, J. J., and de Sande, F. (2012). accULL: An user-directed approach to heterogeneous programming. In *2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications*, pages 654–661.

Rotem, E., Naveh, A., Ananthakrishnan, A., Weissmann, E., and Rajwan, D. (2012). Power-management architecture of the Intel microarchitecture code-named Sandy Bridge. *IEEE Micro*, 32(2):20–27.

Rudy, G., Khan, M. M., Hall, M., Chen, C., and Chame, J. (2011). *A Programming Language Interface to Describe Transformations and Code Generation*, pages 136–150. Springer Berlin Heidelberg, Berlin, Heidelberg.

Sanchez, R. N., Amaral, J. N., Szafron, D., Pirvu, M., and Stoodley, M. (2011). Using machines to learn method-specific compilation strategies. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 257–266, Washington, DC, USA. IEEE Computer Society.

Sher, G. (2011). DXNN: Evolving complex organisms in complex environments using a novel TWEANN system. In *Proceedings of the 13th Annual Conference Companion on Genetic and Evolutionary Computation*, GECCO '11, pages 149–150, New York, NY, USA. ACM.

Sher, G., Martin, K., and Dechev, D. (2014). Preliminary results for neuroevolutionary optimization phase order generation for static compilation. In *Proceedings of the 11th Workshop on Optimizations for DSP and Embedded Systems*, ODES '14, pages 33–40, New York, NY, USA. ACM.

Silvano, C., Sciuto, D., Bruschi, D., and Beltrame, G. (2006). Decision-theoretic exploration of multiprocessor platforms. In *Proceedings of the 4th International Conference on Hardware/-Software Codesign and System Synthesis (CODES+ISSS '06)*, pages 205–210.

Song, L. and Kavi, K. (2004). What can we gain by unfolding loops? *SIGPLAN Not.*, 39(2):26–33.

SPEC. SPEC CPU 2006. https://www.spec.org/cpu2006/ [online].

SPEC. SPEC CPU2000. https://www.spec.org/cpu2000/ [online].

SPEC. SPEC CPU95. https://www.spec.org/cpu95/CFP95/ [online].

SPEC. Specjvm2008 benchmarks. https://www.spec.org/jvm2008/ [online].

SPEC. Specjvm98 benchmarks. https://www.spec.org/jvm98/ [online].

Spinczyk, O., Gal, A., and Schröder-Preikschat, W. (2002). AspectC++: An aspect-oriented extension to the C++ programming language. In *Proceedings of the Fortieth International Conference on Tools Pacific: Objects for Internet, Mobile and Embedded Applications*, CRPIT '02, pages 53–60, Darlinghurst, Australia, Australia. Australian Computer Society, Inc.

Stanley, K. O. and Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evol. Comput.*, 10(2):99–127.

STMicroelectronics. STM32F411RE. http://www.st.com/en/microcontrollers/stm32f411re.html [online].

Suganuma, T., Ogasawara, T., Takeuchi, M., Yasue, T., Kawahito, M., Ishizaki, K., Komatsu, H., and Nakatani, T. (2000). Overview of the IBM Java just-in-time compiler. *IBM Systems Journal*, 39(1):175–193.

Texas Instruments (2008a). TMS320C64x+ DSP image/video processing library (v2.0) programmer's reference (rev. a).

Texas Instruments (2008b). TMS320C64x+ DSP little-endian library programmer's reference (rev. b).

Torczon, L. and Cooper, K. (2011). *Engineering A Compiler*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition.

Touati, S.-A.-A. and Barthou, D. (2006). On the decidability of phase ordering problem in optimizing compilation. In *Proceedings of the 3rd Conference on Computing Frontiers*, CF '06, pages 147–156, New York, NY, USA. ACM.

Vahid, F. and Givargis, T. (2001). *Embedded System Design: A Unified Hardware/Software Introduction*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition.

van Deursen, A., Dinesh, T. B., and van der Meulen, E. A. (1994). *The ASF+SDF Meta-environment*, pages 411–412. Springer London, London.

VeriPool. Verilator. http://www.veripool.org/wiki/verilator [online].

Vuduc, R., Demmel, J. W., and Yelick, K. A. (2005). OSKI: A library of automatically tuned sparse matrix kernels. In *Institute of Physics Publishing*.

Whaley, R. C. and Dongarra, J. J. (1998). Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, SC '98, pages 1–27, Washington, DC, USA. IEEE Computer Society.

Whaley, R. C., Petitet, A., and Dongarra, J. J. (2000). Automated empirical optimization of software and the ATLAS project. *PARALLEL COMPUTING*, 27:2001.

Wicker, L. J. NSSL collaborative model for atmospheric simulation (NCOMMAS). http://www.nssl.noaa.gov/~wicker/commas.html [online].

Wilkes, M. V. (1968). Computers then and now. *J. ACM*, 15(1):1–7.

Xilinx. MicroBlaze soft processor core. http://www.xilinx.com/tools/microblaze.htm [online].

Xiong, J., Johnson, J., Johnson, R., and Padua, D. (2001). SPL: A language and compiler for DSP algorithms. *ACM SIGPLAN Notices - Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, 36(5):298–308.

Yang, H., Gao, G. R., Marquez, A., Cai, G., and Hu, Z. (2000). Power and energy impact by loop transformations. In *In Proceedings of the Workshop on Compilers and Operating Systems for Low Power 2001, Parallel Architecture and Compilation Techniques*.

Yi, Q. (2012). POET: A scripting language for applying parameterized source-to-source program transformations. *Software—Practice & Experience*, 42(6):675–706.

Zhao, J., Nagarakatte, S., Martin, M. M., and Zdancewic, S. (2012). Formalizing the LLVM intermediate representation for verified program transformations. *ACM SIGPLAN Notices - Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 47(1):427–440.